Desenvolupament de jocs per dispositius mòbils

Joan Climent Balaguer

Programació multimèdia i dispositius mòbils

Índex

Introducció **5**

Resultats d'aprenentatge **7**

1 Programació de jocs en Android **9**
1.1 Conceptes bàsics de la programació de jocs . . . 11
1.2 Entrada de dades . . . 14
1.2.1 Pantalla tàctil . . . 14
1.2.2 Acceleròmetre . . . 19
1.2.3 Teclat / Botons . . . 24
1.3 Motors de jocs . . . 27

2 Desenvolupament d'un joc **29**
2.1 LibGDX . . . 29
2.2 Començant amb LibGDX . . . 30
2.3 Desenvolupament del joc . . . 34
2.3.1 Classes del joc . . . 34
2.3.2 Implementació inicial . . . 37
2.3.3 Elements del joc . . . 42
2.3.4 Gràfics i sons . . . 52
2.3.5 Control de la nau . . . 67
2.3.6 Col·lisions . . . 71
2.3.7 Text . . . 77
2.3.8 Preferències . . . 81
2.3.9 Actions . . . 82
2.3.10 Hit . . . 86
2.3.11 Configuració de l'aplicació d'Android . . . 87

Introducció

Avui en dia els videojocs són un dels elements més importants de totes les plataformes, tant al sector dels telèfons mòbils com dels ordinadors de sobretaula i portàtils. El fet de disposar d'una major varietat de jocs pot ser decisiva en l'elecció dels usuaris per una o altra plataforma, i això les empreses que hi ha al darrere dels sistemes operatius ho saben i dediquen gran part del seu esforç a cuidar aquest sector.

En la programació de dispositius mòbils ens trobem amb algunes diferències respecte la programació de jocs per a ordinador o per a consoles. Algunes de les més importants són:

- Les dimensions de la pantalla són reduïdes
- Tenim menys elements d'entrada d'informació i entre ells disposem dels sensors, que no són habituals en ordinadors
- Hem d'adaptar el joc tenint en compte els diferents tipus de dispositius disponibles
- S'han de prioritzar aspectes com el consum energètic o l'espai que ocupa el joc a causa de les característiques dels dispositius mòbils

En l'apartat "Programació de jocs en Android" farem un repàs als conceptes bàsics de la programació de videojocs i analitzarem els principals mètodes d'entrada de què disposem a la plataforma Android: sensors, pantalla tàctil i el teclat.

En l'apartat "Desenvolupament d'un joc" aprendrem a crear un joc des de zero fent servir un motor de jocs: LibGDX. Aprendrem a crear personatges, a assignar-los gràfics, a aplicar-los transformacions, etc.

Per seguir els continguts d'aquesta unitat és molt recomanable anar creant i provant l'aplicació pas a pas, provant les diferents configuracions i mètodes per assegurar-se que s'entén tot el que es va realitzant. Necessitareu també consultar l'ajuda contextual de l'Android Studio i la documentació de LibGDX de manera regular per descobrir noves funcions dels objectes utilitzats.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Selecciona i prova motors de jocs analitzant l'arquitectura de jocs 2D i 3D.

- Descriu els conceptes fonamentals de l'animació 2D i 3D.
- Identifica els elements que componen l'arquitectura d'un joc en 2D i 3D.
- Analitza els components d'un motor de jocs.
- Analitza entorns de desenvolupament de jocs.
- Analitza diferents motors de jocs, les seves característiques i funcionalitats.
- Identifica els blocs funcionals d'un joc existent.
- Defineix i executa els processos de *render*.
- Reconeix la representació lògica i espacial d'una escena gràfica sobre un joc existent.

2. Desenvolupa jocs 2D i 3D senzills fent servir motors de jocs.

- Estableix la lògica d'un nou joc.
- Crea objectes i defineix els fons.
- Instal·la i utilitza extensions per al tractament d'escenes.
- Utilitza instruccions gràfiques per determinar les propietats finals de la superfície d'un objecte o imatge.
- Incorpora so als diferents esdeveniments del joc.
- Desenvolupa i implanta jocs per a dispositius mòbils.
- Realitza proves de funcionament i optimització dels jocs desenvolupats.
- Documenta les fases de disseny i desenvolupament dels jocs creats.

1. Programació de jocs en Android

La programació de videojocs és una de les parts més complexes a dins del món de la informàtica ja que requereix de molts coneixements tècnics que a més comprenen molts sectors. Per tenir-ne una idea bàsica, per crear un joc es necessita:

- Definir el concepte del joc (equip creatiu)
- Escriure el guió del joc (guionistes)
- Dissenyar els personatges, escenaris i nivells (dissenyadors gràfics)
- Crear els efectes sonors, música i veus (equip de so)
- Programar la lògica del joc (programadors)

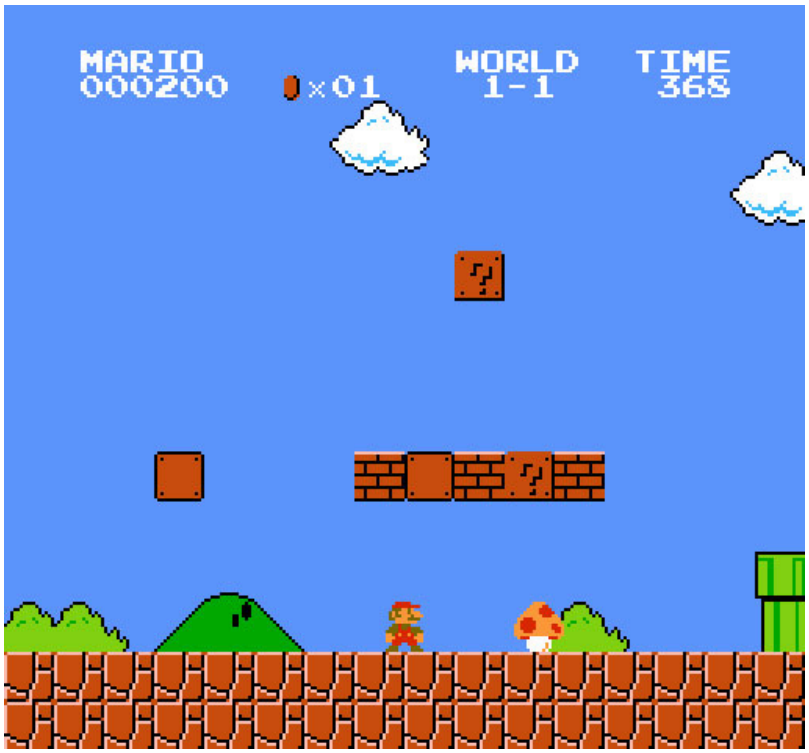
I a més, una vegada estigui tot llest, s'ha de comprovar que tot funcioni correctament per a finalment passar a la postproducció.

Com podem veure, la creació d'un videojoc no és trivial i implica el treball i la coordinació de moltes persones.

En aquesta unitat acabarem desenvolupant un joc, anirem fent poc a poc alguns dels passos descrits a la llista anterior, llavors, abans de posar-nos a programar, haurem de decidir la temàtica del nostre joc. Per fer això necessitarem conèixer alguns dels distints gèneres que existeixen:

- Plataformes: un dels gèneres per excel·lència als jocs en 2 dimensions (2D), molt popular gràcies a jocs com *Donkey Kong* o *Super Mario Bros* (vegeu figura 2.1). En aquest gènere l'objectiu és el d'anar esquivant obstacles i en la majoria dels casos eliminar els enemics que ens anem trobant pel camí. Amb la popularització dels jocs en 3 dimensions (3D), els jocs de plataformes han disminuït respecte altres gèneres però sense arribar a desaparèixer.

FIGURA 1.1. Joc de plataformes: Super Mario Bros



Un *mod* (videojocs) és una modificació d'un joc en el qual fent servir la mateixa base, es canvien els objectius i fins i tot la jugabilitat de l'original.

- Tirs: jocs en el quals l'objectiu principal és atacar els enemics fent servir armes de foc. Entre els seus subgèneres més coneguts trobem:

  – En primer persona: en anglès *First Person Shooter* (*FPS*), són jocs en els quals es visualitza allò que el protagonista veuria a la realitat (perspectiva subjectiva). Aquest tipus de perspectiva combinada amb l'ús d'armes donen nom al gènere. Avui en dia és molt popular als jocs amb opció multijugador, és a dir, en aquells que els usuaris juguen entre ells i no contra personatges controlats per una intel·ligència artificial. Alguns dels jocs més coneguts d'aquest subgènere serien: *Doom*, *Half Life* i el seu *mod Counter Strike* o la saga *Call of duty*.

  – En tercera persona: en anglès *Third Person Shooter* (*TPS*), són molt similars en concepte als *FPS* a excepció que la càmera se situa a la part posterior del personatge, és a dir, veiem totalment o parcialment el cos d'aquest. Alguns exemples serien *Tomb Raider*, les sagues *Resident Evil* o el *Gears of war*.

  – *Shoot 'em up*: jocs en els quals habitualment controlem un vehicle que s'enfronta amb una gran quantitat d'enemics a qui ha de destruir. En aquest subgènere és molt comú trobar perspectives zenitals o laterals i gràfics en 2D. Exemples de jocs d'aquest gènere serien: *Thunder Force*, *R-Type* o *1942*.

- Aventures gràfiques: molt populars durant els anys 90, tenen com a principal objectiu anar avançant en la història que ens proposen bé sigui mitjançant la resolució de trencaclosques, mitjançant diàlegs amb personatges o fent servir objectes. Entre les aventures gràfiques més populars trobem la saga

Monkey Island, *Broken Sword*, *The day of the tentacle* o *The longest journey* entre altres.

- Conducció/carreres: entraria a dins del gènere de simuladors. En aquest tipus de jocs, bé sigui amb una perspectiva en primera o tercera persona, el jugador condueix un vehicle per competir amb altres jugadors i arribar el primer a la meta. Alguns exemples d'aquest gènere serien la saga *Need for speed*, *Gran Turismo* o els distints *Mario Kart*.

Joc casual

Un joc casual és un joc en què la jugabilitat és molt senzilla, no necessita gaire temps de dedicació ni habilitats especials per jugar-lo. Ja existia abans de les plataformes mòbils però amb aquestes ha pres una gran importància.

Molts dels jocs presents avui en dia a les plataformes mòbils són considerats "jocs casuals", aquest tipus de jocs tenen també un gènere associat. Exemples més evidents d'aquest tipus de jocs són els coneguts *Angry Birds* i *Candy Crush Saga*, tots dos tenen una mecànica simple i nivells molt curts que ens permetran jugar tot i no tenir molt de temps disponible, per exemple entre una parada i una altra del metro.

1.1 Conceptes bàsics de la programació de jocs

Abans de començar amb la programació de videojocs hi ha una sèrie de conceptes que caldria conèixer. Fem un repàs d'alguns que ens poden ser útils per entendre el funcionament dels jocs o bé que ens ajudaran en el seu procés de creació.

1. Píxels i resolució de pantalla

Totes les pantalles existents al mercat, incloses les dels telèfons mòbils i tauletes, estan dividides en píxels.

> Un **píxel** és la menor unitat que podem representar en una pantalla. Una pantalla està formada per milions de píxels, dels quals necessitem conèixer-ne la posició respecte aquesta i el color que té.

La **resolució** d'una pantalla ens informarà de la quantitat de divisions horitzontals i verticals de què disposem, i això determinarà el número de píxels que tindrà. Per exemple, quan ens diuen que un telèfon mòbil té una resolució de 1080 x 1920 ens estan informant de la quantitat de divisions que té la pantalla en ample i alt respectivament (mirant el telèfon en *portrait* o les tauletes en *landscape*). Si multipliquem les dues divisions, ens donarà el total de píxels, amb la resolució de l'exemple anterior tindríem un total de 2073600 píxels.

Si a més tenim en compte les dimensions de pantalla, podem calcular la **densitat de píxels** (mesurada en *ppi*, *píxels per inch*) de la pantalla, és a dir, quants píxels hi ha per cada polzada de pantalla; a major densitat, més nítida serà la visualització de la pantalla. Per calcular la densitat de píxels hem d'aplicar la següent fórmula:

$$ppi = \frac{\sqrt{r_h^2 + r_v^2}}{d_i}$$

on r_h és la resolució horitzontal, r_v la vertical i d_i la diagonal de la pantalla, és a dir, les polzades que té. Si sabem que la resolució de l'exemple anterior era d'un dispositiu amb una pantalla de 4,95", la densitat de píxels seria:

$$ppi = \frac{\sqrt{1080^2 + 1920^2}}{4,95} \approx 445ppi$$

.

La profunditat de color fa referència al número de bits que dediquem a representar els colors, a major número de bits, més colors es podran fer servir. Per exemple, amb una profunditat de color de 16 bits es poden representar 2^{16} colors, és a dir, 65536 colors.

2. *Sprites* i animacions

Un ***sprite*** és una imatge en mapa de bits d'un determinat ample i alt. De cada píxel que forma aquesta imatge cal guardar el color que tindrà així que l'espai ocupat per un *sprite* serà el total de píxels de la imatge multiplicat per la seva profunditat de color. Una imatge de 200x200 píxels amb una profunditat de color de 32 bits ocuparia 1.280.000 bits, que passat a bytes i posteriorment a kibibytes farien un total de 156,25 KiB.

Zoòtrop

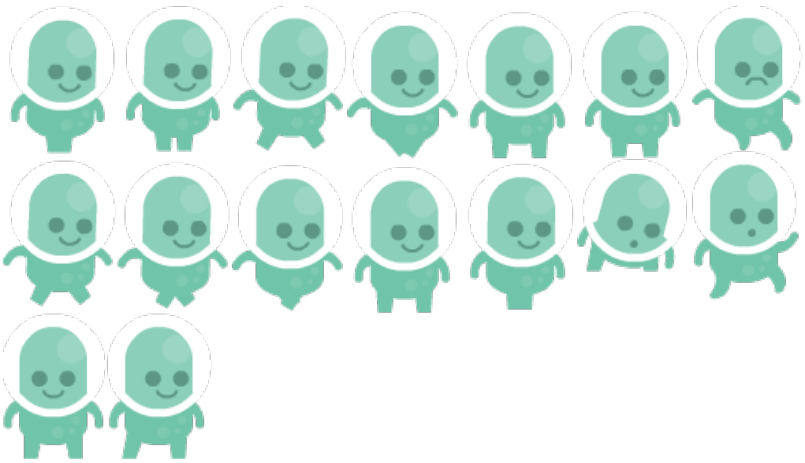
Per tal de reduir la memòria que poden ocupar els *sprites* en ser redimensionats, es fa servir el concepte d'*sprite sheet*: un mapa de bits que conté totes les imatges que es faran servir a la nostra aplicació, del qual traurem tota aquella informació innecessària (com per exemple píxels transparents o colors que són innecessaris). Al final haurem d'aconseguir fer més lleugers els recursos gràfics amb la qual cosa obtindrem diverses millores: reduirem l'espai utilitzat per la nostra aplicació i en millorarem el rendiment. Al següent vídeo podeu veure un exemple d'aquest concepte:

https://www.youtube.com/embed/crrFUYabm6E?controls=1

El zoòtrop va ser un dispositiu que produïa la il·lusió de moviment i que és considerat un dels precursors del cinema. Aquest dispositiu era circular i podíem observar una animació creada a partir de fotogrames a través d'un visor vertical.

Per tal de realitzar animacions, necessitarem crear els *sprites* dels objectes dels diferents estats de l'animació, tal com es feia a un zoòtrop. Afegirem tots els *sprites* a l'*sprite sheet* i després els anirem canviant a la nostra aplicació per donar la sensació de moviment. Un exemple el trobem a la figura 2.3.

FIGURA 1.2. Sprite sheet



3. *Frame rate*

El *frame rate* fa referència a la quantitat d'imatges (*frames*) que es mostren per segon. És una mesura de freqüència i la seva unitat són els *frames per second* (*FPS*).

4. *Polling / Event handling*

El terme *polling* fa referència a una consulta periòdica d'un valor, habitualment obtingut d'un dispositiu de *hardware* (un dispositiu d'entrada o un sensor, per exemple).

La gestió de l'entrada de l'usuari o l'estat dels sensors es pot fer mitjançant *polling* o mitjançant *event handling* (gestió d'esdeveniments).

- Amb *polling* el que fem és comprovar l'estat actual d'un dispositiu d'entrada, per exemple en un bucle del joc comprovem si s'ha pitjat la pantalla o una tecla.

- El sistema *event handling* ens permet gestionar la informació d'entrada a través d'*events* (esdeveniments, en català), és a dir, important, quan la necessitem conèixer, la seqüència dels *events*, per exemple, quan passem el dit per la pantalla ens interessa primer tractar el moment en què el dit toca la pantalla, després tractem el desplaçament del dit i finalment gestionem quan l'usuari aixeca el dit de la pantalla. Serien, per exemple, els *events*: `touchDown`, `touchDragged` i `touchUp`.

5. Gràfics 3D

Per tal de desenvolupar aplicacions que disposin de gràfics en tres dimensions haurem de recórrer a APIs gràfiques. Actualment les principals són:

- **OpenGL**: *Open Graphics Library* és una API multiplataforma i multillenguatge per a la creació de gràfics en 2D i 3D. La versió 4.5 va ser publicada l'11 d'agost de 2014.

- **Direct3D**: API gràfica de Microsoft a dins de les llibreries *DirectX*. Ens permet crear gràfics en 3D per a les plataformes i sistemes operatius de Microsoft: videoconsoles Xbox i sistemes operatius Windows (inclosos els sistemes operatius per a dispositius mòbils).

OpenGL ES (*OpenGL for Embedded Systems*) és una versió reduïda d'OpenGL dissenyada per a dispositius mòbils o consoles de videojocs. La numeració és distinta entre OpenGL i OpenGL ES.

Les versions d'Android Lollipop 5.X disposen de la versió d'OpenGL ES 3.1. Si observem el "*hello world*" d'OpenGL que podem trobar a la pàgina de desenvolupadors http://developer.android.com/training/graphics/opengl/index.html podrem veure que la programació en 3D no és trivial i que requereix de molts coneixements i temps. Al tutorial enllaçat se'ns expliquen els passos per dibuixar figures geomètriques i per afegir-los moviment.

1.2 Entrada de dades

Un dels aspectes més importants dels videojocs és la manera amb què recollim la informació de l'usuari. A banda de la pantalla tàctil i dels teclats i botons físics, els dispositius mòbils acostumen a incorporar molts sensors que ens faciliten aquesta interacció i que estan dividits en tres categories:

- Sensors de moviment: on trobem acceleròmetres, sensors de gravetat, giroscopis, etc.
- Sensors d'entorn: baròmetres, termòmetres i sensors de llum.
- Sensors de posició: sensors d'orientació i magnetòmetres.

Cadascun dels sensors ens podrà ser útil segons la finalitat de la nostra aplicació però sempre haurem de tenir en compte que no tots els dispositius incorporen tots els sensors. Android ens facilitarà l'accés a la informació sobre quins sensors tenim disponibles, les característiques de cadascun d'ells i l'obtenció de la informació que ens proporcionen a través d'un *framework* propi que forma part del *package* `android.hardware`.

1.2.1 Pantalla tàctil

La pantalla tàctil és el dispositiu d'entrada d'informació per excel·lència als dispositius mòbils. A través d'ella interactuem amb el dispositiu la major part del temps.

Des de la perspectiva de la programació d'aplicacions, per controlar què ha fet l'usuari, disposem del mètode `onTouchEvent(MotionEvent event)` de la classe `Activity` o bé del mètode `onTouch(View v, MotionEvent event)` de la interfície `OnTouchListener`. La diferència entre els dos és: mentre que al mètode `onTouchEvent` l'aplicació registrarà totes les pulsacions que es produeixin en la nostra activitat, independentment del *view* sobre el qual estem polsant, implementant la interfície `onTouchListerner` podrem especificar de quins *views* ens interessa conèixer les pulsacions, fent el corresponent `setOnTouchListener`.

Podeu descarregar el projecte de la següent aplicació a l'apartat "Touch Events" dels annexos.

Per entendre els *events* que es produeixen cada cop que l'usuari interactua amb la pantalla, crearem una aplicació, tot iniciant un nou projecte de nom **TouchEvents** i de domini **cat.xtec.ioc**.

El *layout* de la nostra aplicació tindrà una capsa de text que mostrarà els *events* que s'aniran generant (i que ens permetrà fer *scroll*), un *LinearLayout* que controlarà les accions de l'usuari amb la pantalla tàctil i una altra capsa de text que mostrarà la direcció en la qual l'usuari ha arrossegat el dit. Si posem aquests elements a dins d'un *LinearLayout* ens quedarà el següent codi:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">

    <ScrollView
        android:id="@+id/scroll"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1"
        android:fillViewport="true"
        android:scrollbars="vertical">

        <TextView
            android:id="@+id/accions"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:scrollbars="vertical"
            android:text=""
            android:textAppearance="?android:attr/textAppearanceSmall" />
    </ScrollView>

    <LinearLayout
        android:id="@+id/linear"
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1"
        android:background="#ff88eeff"
        android:orientation="vertical">


        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:text=""
            android:id="@+id/direccio" />
    </LinearLayout>

</LinearLayout>
```

Observeu que el *textView* d'identificador "accions" és a dins de l'element *ScrollView*. Això ens permetrà fer *scroll* en cas de produir-se *overflow*, és a dir, que el text no càpiga a l'espai assignat.

Cada cop que l'usuari faci una pulsació o arrossegui el dit pel *LinearLayout* d'identificador "linear", es generaran una sèrie d'*events* que quedaran recollits al *textView* "accions".

Abans de passar a la part del codi, modificarem l'entrada de menú *settings* per donar-li una utilitat. Aquesta netejarà el contingut del *textView* "accions". Editem el fitxer */res/menu/menu_main.xml* per deixar-lo de la següent manera:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
    <item android:id="@+id/action_clear" android:title="@string/action_clear"
  android:orderInCategory="100" app:showAsAction = "ifRoom"/>
</menu>
```

Hem fer tres modificacions de l'element *item*:

- `android:id=''@+id/action_settings''` per `android:id=''@+id/action_clear''`: canviem el nom de l'identificador.
- `android:title=''@string/action_settings''` per `android:title=''@string/action_clear''`: canviem el text que es mostrarà a la pantalla.
- i app:showAsAction = "never" per app:showAsAction = "ifRoom": fem que l'element de menú sigui visible a l'`ActionBar` sempre que hi hagi lloc disponible.

Com que hem canviat l'*string* que es mostrarà per pantalla, haurem d'incorporar-la al fitxer */res/values/strings.xml* i podrem eliminar els recursos que no es facin servir, quedant de la següent manera:

```
<resources>
    <string name="app_name">TouchEvents</string>

    <string name="action_clear">Neteja</string>
</resources>
```

Una vegada ja tenim les vistes i el menú preparats, donem funcionalitat a la nostra aplicació. Com que volem enregistrar les pulsacions a un *view* concret haurem d'implementar la interfície `OnTouchListener` i per això, haurem d'afegir el corresponent `implements` en la definició de la classe:

```
public class MainActivity extends ActionBarActivity implements View.
    OnTouchListener {
```

Definirem les variables corresponents als *views* que volem modificar i afegirem dos *float* que guardaran l'inici d'un *event* d'arrossegar (*drag*): "iniciX" i "iniciY". Per últim necessitarem un enter ("numAccio") que actuarà com a comptador dels *events* que s'han generat.

Codi de definició de les variables:

```
TextView accions, direccio;
    float iniciX, iniciY;
    int numAccio;
```

Al mètode `onCreate`, inicialitzarem les variables i afegirem l'`OnTouchListener` al *textView* d'accions, que quedarà de la següent manera:

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Afegim el "Listener" dels events de "touch" sobre el layout inferior
            .
        LinearLayout linear = (LinearLayout) findViewById(R.id.linear);
        linear.setOnTouchListener(this);

        // Inicialitzem les variables
```

```
        accions = (TextView) findViewById(R.id.accions);
        direccio = (TextView) findViewById(R.id.direccio);
        iniciX = iniciY = 0;
        numAccio = 0;

    }
```

Pel que fa al menú, deixem la funció `onCreateOptionsMenu` sense modificar i controlem si l'usuari ha fet clic a l'opció "Neteja" des de la funció `onOptionsItemSelected`, si l'usuari vol esborrar: netejarem el *textView* i posarem el comptador a 0:

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //Netegem el TextView i posem el comptador a 0
        if (id == R.id.action_clear) {
            accions.setText("");
            numAccio = 0;
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
```

Tota la gestió de les pulsacions de l'usuari la farem a la funció `onTouch(View v, MotionEvent event)` en la qual disposem de dos arguments: la vista sobre la que s'ha fet una pulsació i el `MotionEvent` generat, que entre d'altra informació, ens permetrà saber quin *event* s'ha realitzat, sent alguns dels més importants els següents:

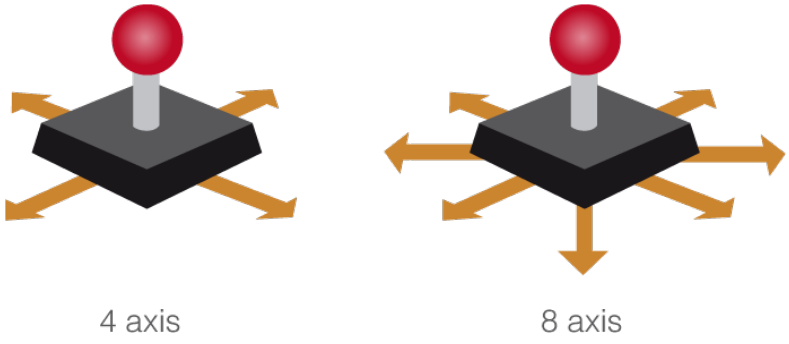
- **ACTION_DOWN**: es produeix quan l'usuari inicia una pulsació, és a dir, al moment exacte que el dit fa contacte en la pantalla.
- **ACTION_UP**: quan finalitzem la pulsació, quan el dit deixa de tocar la pantalla.
- **ACTION_MOVE**: entre els *events* ACTION_DOWN i ACTION_UP hi ha hagut algun canvi en la posició del dit, si hi ha moviment és perquè l'usuari ha desplaçat el dit per la pantalla durant la pulsació.
- **ACTION_POINTER_DOWN**: s'ha produït una pulsació d'un dit que no és el principal (passem a generar *events multi touch*). Quan aquest dit s'aixeca de la pantalla genera un **ACTION_POINTER_UP**.
- **ACTION_POINTER_INDEX_MASK**: conjuntament amb **ACTION_POINTER_INDEX_SHIFT** ens permetrà saber quin índex té la pulsació en un *event multi touch*.

Per controlar un *event* d'arrossegament de dit farem servir els *events* `ACTION_DOWN` (on guardarem la posició inicial del dit a la pantalla) i `ACTION_UP` (que guardarà la posició final i ens permetrà calcular la direcció). Si volem

saber quin desplaçament ha fet l'usuari, haurem de descartar tota la informació innecessària i calcular la trajectòria segons ens interessi; en *4-axis* o en *8-axis* (vegeu figura 2.4).

La coordenada (0,0) de la pantalla se situa a la part superior esquerra del dispositiu.

FIGURA 1.3. Moviments en 4 i 8 axis



- **4-axis**: els valors possibles són dalt, dreta, baix i esquerra. Per saber la direcció en què l'usuari ha arrossegat el dit haurem de veure en quin eix (horitzontal o vertical) i en quina direcció ho ha fet. Com que és molt complex realitzar un desplaçament perfecte en un únic eix, haurem de descartar la informació que ens arriba de l'altre; una manera de fer-ho és detectar a quin eix s'ha produït un major desplaçament fent la resta entre els valors absoluts dels punts inicials i els punts finals. Per realitzar aquesta implementació haurem de guardar els punts X i Y inicials quan es produeix l'*event* `ACTION_DOWN` i recollir les posicions finals a l'`ACTION_UP`. Una vegada sabem l'eix, podrem obtenir fàcilment el sentit fent la resta entre el punt inicial i el final. Aquesta serà la implementació que farem a la nostra aplicació.

- **8-axis**: a més les direccions que disposem amb 4-axis, podem considerar les que es produeixen a partir d'una combinació dels dos eixos, és a dir, les posicions: dalt-dreta, baix-dreta, baix-esquerra i dalt-esquerra. Com que és molt complex realitzar un desplaçament de "0" en qualsevol eix, per tal d'implementar aquesta solució haurem de definir un valor mínim a partir del qual es considera que hi ha hagut un desplaçament en una determinada direcció. Si no afegim aquest valor mínim, ens trobaríem amb què tots els desplaçaments serien una combinació dels dos eixos i no hi hauria cap moviment dels 4 *axis* principals.

El codi de la funció `onTouch` serà el següent:

```
1  @Override
2      public boolean onTouch(View v, MotionEvent event) {
3
4          if (event.getAction() == MotionEvent.ACTION_DOWN) {
5
6              // Registrem l'event al TextView
7              accions.setText(String.valueOf(numAccio) + " - " + "ACTION_DOWN\n"
                   + accions.getText());
8              iniciX = event.getX();
9              iniciY = event.getY();
10
```

```
11
12          } else if (event.getAction() == MotionEvent.ACTION_UP) {
13
14              // Registrem l'event al TextView
15              accions.setText(String.valueOf(numAccio) + " - " + "ACTION_UP\n" +
                    accions.getText());
16              float finalX = event.getX();
17              float finalY = event.getY();
18
19              //Comprovem quin dels dos moviments ha estat major. Si és l'
                    horitzontal
20              if (Math.abs(finalX - iniciX) > Math.abs(finalY - iniciY)) {
21
22                  // Si la X final és major que la inicial ha fet "drag" cap a la
                        dreta
23                  if (finalX > iniciX) {
24                      direccio.setText("→");
25                  // Si no, cap a l'esquerra
26                  } else {
27                      direccio.setText("←");
28                  }
29              // Si és el vertical
30              } else {
31                  // Si la Y final és major que la inicial ha fet "drag" cap
                        avall
32                  if (finalY > iniciY) {
33                      direccio.setText("↓");
34                  // Si no, cap a dalt
35                  } else {
36                      direccio.setText("↑");
37                  }
38              }
39
40          } else if (event.getAction() == MotionEvent.ACTION_MOVE) {
41
42              // Registrem l'event al TextView
43              accions.setText(String.valueOf(numAccio) + " - " + "ACTION_MOVE\n"
                    + accions.getText());
44
45          }
46          // Incrementem el comptador d'acció
47          numAccio += 1;
48          return true;
49      }
```

Si l'acció realitzada (`event.getAction()`) ha estat `ACTION_DOWN` guardarem les coordenades X i Y de la pulsació i quan es produeixi `ACTION_UP` recollirem els valors finals.

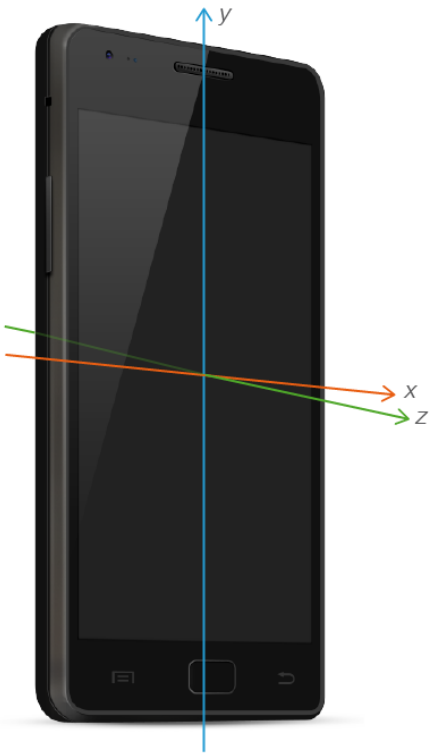
Com que controlarem els moviments en *4-axis* haurem de comprovar quin ha estat l'eix que més s'ha deplaçat amb la condició `Math.abs(finalX - iniciX) > Math.abs(finalY - iniciY)`, és a dir, mirem quina és la diferència més gran entre els punts inicials i finals a l'eix de les X i l'eix de les Y. Només ens quedarà saber quina ha estat la direcció comparant els punts final i inicial de l'eix.

1.2.2 Acceleròmetre

L'acceleròmetre és uns dels sensors que més informació ens dóna sobre l'usuari. Ens permet conèixer en tot moment en quina posició es troba el mòbil, quins

desplaçaments es produeixen i a quina velocitat. Aquest sensor és molt utilitzat als jocs (per controlar els girs o desplaçaments dels personatges) però també és utilitzat per a accions tan bàsiques com la de girar la pantalla quan l'usuari gira el telèfon. Podem controlar els canvis que es produeixen als eixos X, Y i Z tal com es pot veure a la figura 2.6

FIGURA 1.4. Eixos X, Y i Z en un dispositiu mòbil



A la plataforma Android, els sensors s'han d'enregistrar per poder-se fer servir i els hem d'alliberar quan no siguin necessaris (per evitar consums excessius de bateria). Els mètodes `onPause()` i `onResume()` són els més adequats per realitzar aquestes tasques: quan s'executa l'`onPause()` deixem de seguir els canvis del sensor i continuem llegint-los en tornar a l'aplicació (`onResume()`).

Podeu descarregar el projecte de la següent aplicació a l'apartat "MotionSensor" dels annexos.

Per tal d'introduir els conceptes bàsics de la gestió de l'acceleròmetre desenvoluparem una aplicació que ens mostrarà en tot moment l'estat dels eixos. Creeu un projecte anomenat **MotionSensor** amb el *Company Domain* **cat.xtec.ioc** i creeu una activitat buida deixant els valors per defecte.

El *layout* de la nostra aplicació estarà format per sis *TextView*, tres seran les etiquetes "X", "Y" i "Z" i els altres tres seran els valors dels sensors en cada moment.

El codi XML podria ser:

```
1 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     xmlns:tools="http://schemas.android.com/tools" android:layout_width="
          match_parent"
3     android:layout_height="match_parent" android:paddingLeft="@dimen/
          activity_horizontal_margin"
4     android:paddingRight="@dimen/activity_horizontal_margin"
```

```
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=".
        MainActivity">

    <TextView android:text="X: " android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textX" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:id="@+id/valueX"
        android:layout_alignParentTop="true"
        android:layout_toRightOf="@+id/textX"
        android:layout_toEndOf="@+id/textX" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="Y: "
        android:id="@+id/textY"
        android:layout_below="@+id/textX"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:id="@+id/valueY"
        android:layout_below="@+id/valueX"
        android:layout_alignLeft="@+id/valueX"
        android:layout_alignStart="@+id/valueX" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:text="Z: "
        android:id="@+id/textZ"
        android:layout_below="@+id/textY"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:id="@+id/valueZ"
        android:layout_below="@+id/textY"
        android:layout_toRightOf="@+id/textY"
        android:layout_toEndOf="@+id/textY" />

</RelativeLayout>
```

Pel que fa al codi de l'aplicació, el primer que hem de fer per controlar els canvis al sensor és implementar la classe `SensorEventListener` afegint a la capçalera:

```
public class MainActivity extends ActionBarActivity implements
    SensorEventListener {
```

Una vegada hem afegit l'`implements`, l'Android Studio ens informarà d'un error que solucionarem fent clic a *Implement methods*. Al final de la classe tindrem els mètodes:

```
@Override
    public void onSensorChanged(SensorEvent sensorEvent) {

    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int i) {

    }
```

Aquests mètodes seran cridats quan es produeixin canvis als valors dels sensors (`onSensorChanged`) o en la seva precisió (`onAccuracyChanged`).

En el nostre cas ens interessen els valors dels sensors, ho sigui que haurem d'afegir el codi corresponent per recollir els canvis produïts. Prepararem, però, abans les variables i inicialitzarem el sensor.

Afegirem les següents declaracions de variables just entre la declaració de la classe i el mètode `onCreate`:

```
TextView valueX, valueY, valueZ;
    SensorManager sensorMgr;
    Sensor sensor;
```

per després inicialitzar-les al mètode `onCreate`, els tres *TextView* ens permetran representar per pantalla l'estat dels sensors, el *SensorManager* ens permetrà assignar i enregistrar l'acceleròmetre, al qual podrem accedir a través de la variable *Sensor*.

Així, el contingut del mètode quedarà de la següent manera:

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        valueX = (TextView) findViewById(R.id.valueX);
        valueY = (TextView) findViewById(R.id.valueY);
        valueZ = (TextView) findViewById(R.id.valueZ);

        sensorMgr = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        sensor = sensorMgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sensorMgr.registerListener(this, sensor , SensorManager.
            SENSOR_DELAY_NORMAL);

    }
```

Per tal de deixar de controlar l'activitat dels sensors quan no estem a l'aplicació i evitar així els problemes de bateria descrits anteriorment, hem d'afegir el següent codi a la nostra activitat:

```
@Override
    protected void onPause() {
        super.onPause();
        sensorMgr.unregisterListener(this);
    }

    @Override
    protected void onResume() {
        super.onResume();
```

```
10          sensorMgr.registerListener(this,sensor, SensorManager.
                SENSOR_DELAY_NORMAL);
11      }
```

Quan la nostra activitat no estigui en primer pla (`onPause()`) deixem de seguir els canvis produïts al sensor (`sensorMgr.unregisterListener(this);`) i el tornem a enregistrar quan l'usuari torna a l'aplicació (`onResume()`).

Ja sols ens quedarà escriure el codi del mètode `onSensorChanged(SensorEvent sensorEvent)`, que actualitzarà els *TextView* amb la informació que ens proporciona el sensor.

El paràmetre `sensorEvent` ens permetrà accedir al sensor, la precisió d'aquest, els valors, etc. Així, el primer que comprovarem és si la informació que estem rebent és la del sensor correcte amb el condicional:

```
1   if(sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
```

És a dir, accedim al sensor i fem la crida al mètode `getType()`, si és l'acceleròmetre, llavors actualitzarem els valors dels *TextView*. Per llegir els valors hem de recollir la variable *values*, un *array* que conté la informació de l'eix de les "X", de les "Y" i de les "Z" a les posicions 0, 1 i 2 respectivament. Així, si consultem el `values[0]` obtindrem el valor de la "X".

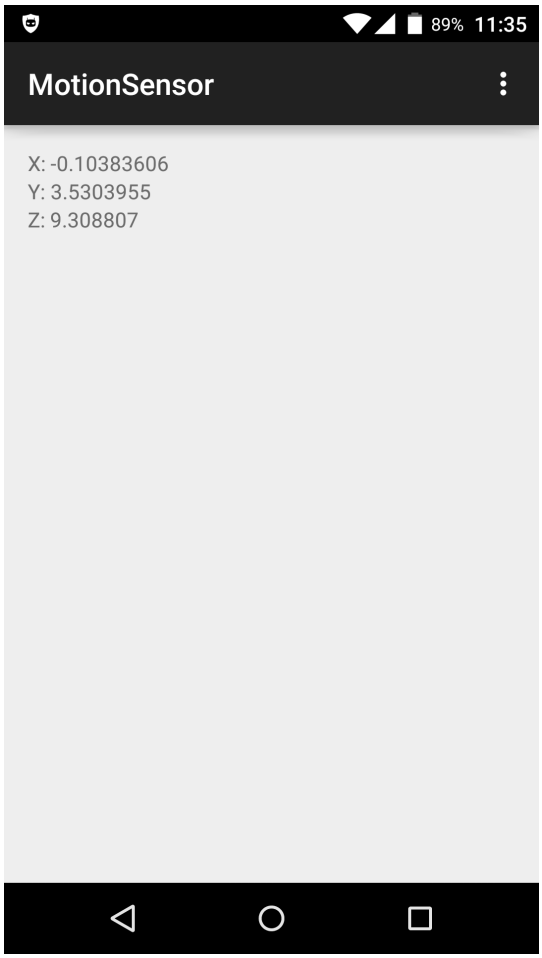
Una vegada comprovat que el sensor és el correcte, assignem als *TextView* els valors dels 3 eixos, passant-los prèviament a *string*.

```
1   @Override
2       public void onSensorChanged(SensorEvent sensorEvent) {
3
4           if(sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
5
6               valueX.setText(String.valueOf(sensorEvent.values[0]));
7               valueY.setText(String.valueOf(sensorEvent.values[1]));
8               valueZ.setText(String.valueOf(sensorEvent.values[2]));
9
10          }
11
12      }
```

Per tal de provar aquesta aplicació necessitarem fer-ho en un telèfon real ja que l'emulador d'Android no disposa d'aquesta opció. En cas de no tenir un dispositiu on provar-la, podeu mirar emuladors alternatius a l'oficial o instal·lar SensorSimulator i configurar-lo perquè treballi amb l'emulador.

El resultat serà el que podeu veure a la figura 2.7. Proveu a girar el telèfon en els distints eixos i observeu els canvis que es produeixen en els valors.

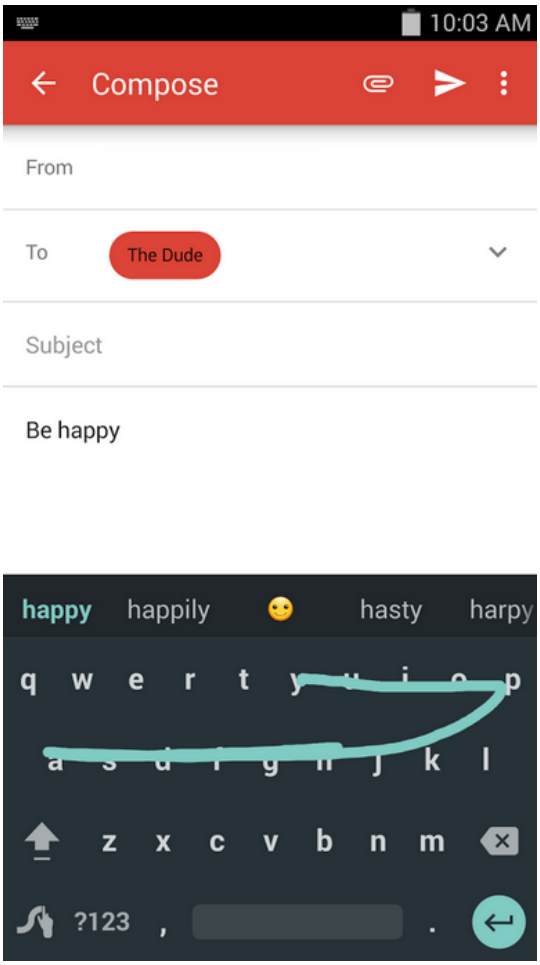
FIGURA 1.5. Projecte MotionSensor en funcionament



1.2.3 Teclat / Botons

Cada vegada és menys freqüent trobar dispositius mòbils amb teclat físic, els teclats virtuals (com per exemple el que veiem a la figura 2.8) han millorat molt pel que fa a entrada de text i prediccions i han substituït pràcticament els teclats físics, que disposen de limitacions quant a funcionalitats i no es poden adaptar al context de les aplicacions tal com fan els teclats en pantalla. No obstant això, no hem de deixar de banda la gestió dels *events* de tecles, a banda del reduït però present sector de dispositius amb teclat incorporat, hem de tenir en compte aquells portàtils amb sistema operatiu Android i més pensant en l'actual tendència de convergència entre sistemes operatius d'escriptori i sistemes operatius pensats per a dispositius mòbils.

FIGURA 1.6. Swype: exemple de teclat virtual en Android



Si volem gestionar aquest tipus d'*events*, de la mateixa manera que amb els de la pantalla tàctil, ho podem fer de dues maneres: gestionant qualsevol *event* que es produeixi a la nostra activitat o controlant aquells *events* dels *views* que nosaltres desitgem.

Gestió des de l'activitat

Per tal de gestionar els *events* de l'activitat, disposem de les funcions:

- *onKeyDown*: cridat en el moment de polsar la tecla.
- *onKeyUp*: cridat en finalitzar un *onKeyDown*, és a dir, quan deixem de pressionar la tecla.
- *onKeyLongPress*: s'executa quan deixem pressionada una tecla.
- *onKeyMultiple*: quan la mateixa tecla produeix diversos *events down* i *up* en un temps reduït.

Un exemple per controlar quan l'usuari prem la tecla de baixar volum seria el següent:

```
@Override
    public boolean onKeyUp(int keyCode, KeyEvent event) {

        if(keyCode == KeyEvent.KEYCODE_VOLUME_DOWN) {

            // Codi de baixar volum
            Toast.makeText(this, "Baixem volum!", Toast.LENGTH_SHORT).show();
            return true;

        }

        return super.onKeyUp(keyCode, event);
    }
```

El mètode `onKeyUp` s'executarà quan deixem de prémer la tecla de volum i rep com a paràmetres:

- `int keyCode`: enter que fa referència a constants definides a la classe KeyEvent. Ens indica quina tecla ha estat premuda.
- `KeyEvent event`: informació sobre l'*event* generat.

A l'exemple anterior comprovem que la tecla que s'ha pressionat és la corresponent a la variable KeyEvent.KEYCODE_VOLUME_DOWN, en cas de ser així controlem l'*event* i hem de retornar `true` en cas de voler gestionar-lo o `false` en cas de voler que sigui gestionat pel següent receptor.

Teniu una llista completa de les constants a http://developer.android.com/reference/android/view/KeyEvent.html així com també els mètodes i funcions disponibles de la classe `KeyEvent`.

Gestió d'un view en concret

Si la gestió dels *events* de tecles l'hem de fer des d'un *view* concret haurem de fer el corresponent *implements* de la classe:

```
public class MainActivity extends ActionBarActivity implements View.
    OnKeyListener {
```

Com sempre ens donarà error i haurem de seleccionar el suggeriment de l'Android Studio i escollir l'opció `Implement methods` que ens afegirà el codi del mètode `onKey(View view, int i, KeyEvent keyEvent)`. Els paràmetres són els mateixos que a la funció `onKeyUp` però a més ens informa del *view* que ha generat l'*event* amb el paràmetre `View view`.

Recordeu que per tal de controlar els *events* d'un *view*, a banda de fer l'*implements* hem d'assignar-li el *listener* amb `vista.setOnKeyListener(this);` on vista serà el *view* corresponent.

1.3 Motors de jocs

Els motors de jocs són *frameworks* dissenyats per a la creació de videojocs, és a dir, proporcionen recursos i metodologies per tal de facilitar-ne la programació. Un motor de joc ens ha de proporcionar una estructura sobre la qual començar a programar i crear la base del nostre joc d'una manera ràpida i senzilla. Aquest tipus de programari acostuma a ser modular, de manera que podem afegir aquelles funcionalitats que necessitem o bé programar-ne de noves sense afectar a la resta de components utilitzats. Alguns dels mòduls que podem trobar són: Tractament de físiques, tractament de col·lisions, intel·ligència artificial, gestió de recursos, etc.

> En general, i sempre que sigui possible, es recomana la utilització d'un motor de joc, d'aquesta manera reduirem el temps de dedicació i els possibles errors de disseny que podria implicar crear un projecte des de l'inici.

El sector dels dispositius mòbils ha experimentat un fort creixement els darrers anys i els sector dels videojocs ha estat un dels més importants. Si analitzem els motors de jocs disponibles trobarem una gran varietat pel que fa a llicències, llenguatges de programació, funcionalitats, plataformes, etc. Fem un repàs d'alguns dels motors de jocs disponibles actualment:

- Unity: desenvolupament de jocs multiplataforma (Android, iOS, Linux, OSX, PS4, Xbox One, etc.). Disposa d'una versió personal i una professional de pagament.
- Unreal engine: motor multiplataforma molt popular de la companyia *Epic Games*. La primera versió del motor es va implementar als *FPS*: *Unreal* i *Unreal Tournament*. És gratuït però s'ha de pagar un *royaltie* d'un 5% en cas de comercialitzar el producte i obtenir beneficis.
- AndEngine: motor conegut en el món dels jocs 2D en Android. És *open source* i gratuït.
- LibGDX: ens permet crear jocs 2D i 3D multiplataforma per a: Windows, Linux, OSX, Android, iOS, BlackBerry i HTML5. És *open source*, gratuït i es programa amb el llenguage Java.
- Godot Game Engine: motor multiplataforma i *open source*, disposa d'un llenguatge propi molt similar a *python*.

Per a la creació del joc en aquesta unitat hem decidit escollir **LigGDX** per les seves característiques i per tenir com a base el llenguatge de programació Java.

2. Desenvolupament d'un joc

Crear un joc partint d'un projecte buit pot ser una tasca molt complexa i que suposarà un repte durant tot el seu desenvolupament. Un error de disseny pot implicar grans canvis fins i tot en etapes inicials del desenvolupament. Per evitar aquests errors i simplificar aquesta tasca (tant en temps com en complexitat) el més adequat és fer servir eines que ja existeixin i que ens garanteixin que els components principals del joc ja estan implementats i que no ens n'haurem de preocupar.

Una vegada ja tenim clar que hem de recórrer a eines de tercers, haurem d'escollir la més adequada. La gran comunitat que hi ha al darrere, la documentació disponible i les tecnologies utilitzades (Java per a la programació i varietat de *plugins* de tercers disponibles) han fet que el motor de jocs escollit per aquest mòdul sigui **LibGDX**.

2.1 LibGDX

LibGDX és un *framework open source* (llicència Apache 2.0) per desenvolupar jocs en 2D i en 3D que ens permetrà generar fàcilment els fitxers binaris per poder-se executar, entre altres, a plataformes com ara:

- Android
- OSX i iOS
- Windows
- GNU/Linux
- HTML5
- BlackBerry

Disposarem, per tant, d'una part de programació comuna a totes les plataformes i una part específica en què configurarem la nostra aplicació per tal que s'executi correctament a la plataforma destí. A la pàgina oficial: http://libgdx.badlogicgames.com/ trobarem les novetats (*changelogs*) de cada versió, les característiques principals del *framework*, els *plugins* disponibles, etc. I especialment, com a programadors, ens interessaran les seccions de documentació i de comunitat, on podrem resoldre la major part dels nostres dubtes.

Estem davant d'un motor de jocs molt complet, que té una gran comunitat al darrere i que ens permetrà realitzar jocs de tot tipus amb resultats professionals sense haver de realitzar una despesa econòmica per utilitzar-lo.

Per seguir les explicacions, i per tal de tenir un exemple pràctic dels conceptes, desenvoluparem un joc de naus espacials: l'**SpaceRace**, on prendrem el rol del capità d'una nau espacial que haurem de dirigir per tal d'evitar col·lidir amb els asteroides que ens trobarem pel camí.

2.2 Començant amb LibGDX

LibGDX compta amb un assistent de configuració de projectes en el qual definirem quines propietats tindrà el nostre projecte. Aquest assistent s'encarregarà de descarregar tot allò necessari i de generar els fitxers que ens permetran importar el projecte al nostre IDE.

Per descarregar l'assistent, accedirem a la secció *Download* de la pàgina oficial: http://libgdx.badlogicgames.com/download.html. El fitxer descarregat serà el **gdx-setup.jar**.

Per executar-lo haurem d'obrir una terminal i executar la comanda `java -jar /ruta/al/fitxer/gdx-setup.jar`, per exemple:

```
1 java –jar ~/Descargas/gdx–setup.jar
```

Se'ns mostrarà una imatge com la que podem veure a la figura 2.1.

FIGURA 2.1. Assitent de projecte de LibGDX

L'assistent ens demana els següents atributs:

- *Name*: nom del projecte, en el nostre cas **spacerace**.
- *Package*: l'espai de noms de la nostra aplicació, per exemple **cat.xtec.ioc**.
- *Game class*: classe principal, serà del tipus `ApplicationListener`. La nostra classe s'anomenarà **SpaceRace**.
- *Destination*: directori al nostre sistema de fitxers on generarà el projecte.
- *Android SDK*: indiquem la ruta a l'SDK en cas que l'Android Studio no el tingui disponible.
- *LibGDX Version*: versió de libGDX, deixeu la proposada.
- *Sub projects*: indiquem les plataformes sobre les quals podrem executar el projecte i que necessiten configuració addicional. Deixem marcades **Desktop** i **Android**.
- *Extensions*: afegits al *framework* per a la gestió de col·lisions, intel·ligència artificial, fonts, etc. Podem deixar marcat **Box2d** (llibreria de físiques 2D) tal com ens proposa.
- *Third party Extensions*: extensions externes a LibGDX. No farem servir cap d'aquestes extensions.
- *Advanced*: opcions avançades. Podrem afegir *mirrors* per a les descàrregues, generació de fitxers per a IDEA o Eclipse o bé el mode *offline*. Ens serà molt útil marcar IDEA ja que l'Android Studio està basat en aquest IDE.

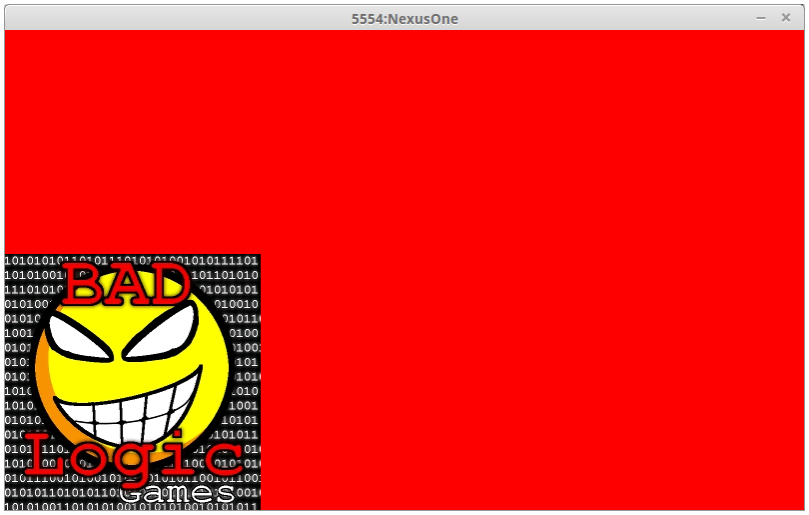
Una vegada emplenats tots els atributs, farem clic a *Generate* i començarà la descàrrega i configuració dels fitxers necessaris, si ens pregunta si volem continuar amb versions més recents de les *android build tools* o de les *Android API*, responeu que sí. Esperarem fins que ens surti el missatge: **BUILD SUCCESSFUL** i la informació sobre com importar el projecte. Podeu veure a la figura 2.2 l'assistent amb la configuració i el resultat final.

FIGURA 2.2. Assistent amb la configuració

Obriu l'Android Studio, feu clic a *File/Open* i seleccioneu el projecte a la ruta indicada a l'assistent. Una vegada obert, si feu clic a la pestanya *Project* de l'esquerra, podreu observar tres subprojectes: *android*, *core* i *desktop*; *android* i *desktop* ens serviran per configurar i personalitzar l'aplicació per a cadascuna de les dues plataformes i a *core* definirem els fitxers comuns, tota la implementació del joc.

Si executeu el projecte en un emulador, obtindreu el resultat que es mostra a la figura 2.3.

FIGURA 2.3. Aplicació executant-se a l'emulador



El logotip de *Bad Logic Games* sobre un fons vermell. Si obrim el fitxer SpaceRace.java del subprojecte *core* trobarem el següent codi:

```
public class SpaceRace extends ApplicationAdapter {
  SpriteBatch batch;
  Texture img;

  @Override
  public void create () {
    batch = new SpriteBatch();
    img = new Texture("badlogic.jpg");
  }

  @Override
  public void render () {
    Gdx.gl.glClearColor(1, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.begin();
    batch.draw(img, 0, 0);
    batch.end();
  }
}
```

L'`SpriteBatch` és l'encarregat de dibuixar objectes en 2D: genera un lot de comandes gràfiques i les optimitza per enviar-les a la GPU. El nostre `SpriteBatch` s'inicialitza al mètode `create()` i dibuixarà tot allò que estigui entre els mètodes `begin()` i `end()`, en aquest cas dibuixarà la imatge `img` i ho farà a la coordenada (0,0). A l'exemple la coordenada (0,0) representa la part inferior esquerra de la pantalla (sistema de coordenades *Y-Up*) però compte perquè no sempre és així, habitualment i al nostre joc, la coordenada (0,0) farà referència a la cantonada superior esquerra (sistema *Y-Down*).

El codi:

```
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

Definirà el color del fons.

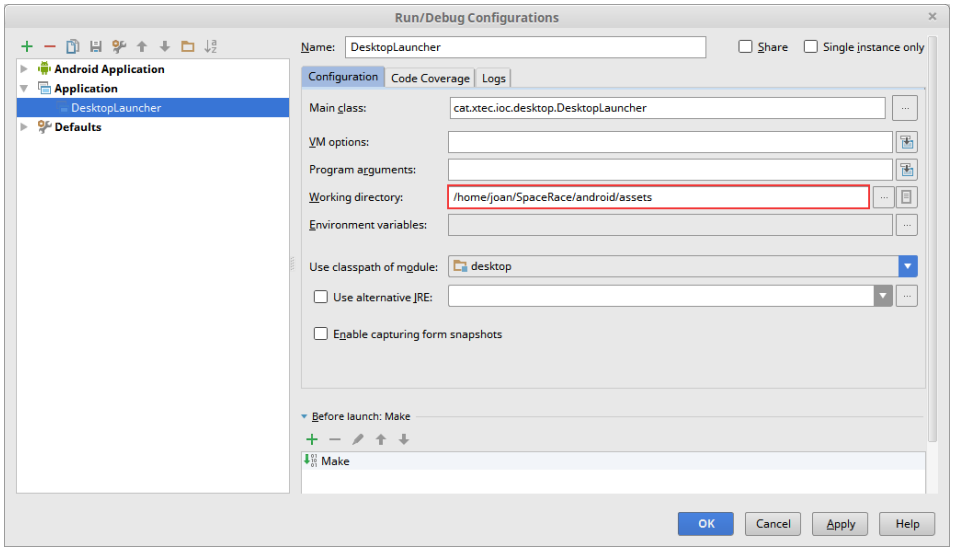
El mètode `glClearColor()` rep 4 paràmetres (del tipus `float` del 0.0 a l'1.0) que fan referència a la quantitat de vermell, de verd, de blau i a l'opacitat respectivament. Un valor 0.0 a l'opacitat indica que l'element és transparent i un 1.0 que és completament opac.

Al codi anterior, carrega el color vermell al *buffer* i el mètode `glClear(GL20.GL_COLOR_BUFFER_BIT)` indica que el *buffer* està habilitat per escriure colors.

El fitxer **badlogic.jpg** el podreu trobar al directori *assets* del subprojecte *android*, en aquest directori deixarem tots els fitxers gràfics que farem servir a l'aplicació i els enllaçarem des de la resta de plataformes. Si fem clic al botó dret sobre el fitxer *DesktopLauncher* del subprojecte *desktop* i seleccionem *Run DesktopLauncher main()* ens saltarà la següent excepció: `Exception in thread "LWJGL Application" com.badlogic.gdx.utils.GdxRuntimeException: Couldn't load file: badlogic.jpg`. Per solucionar-ho, seleccionem l'entrada del menú

Run/Edit Configurations..., seleccionem *Application/DesktopLauncher* i canviem el *working directori* a la ruta d'*assets* d'android (figura 2.4).

FIGURA 2.4. Configuració DesktopLauncher



Si torneu a executar el projecte se us obrirà una finestra amb el mateix contingut que abans: el logotip sobre un fons vermell. Ens serà molt útil poder treballar únicament amb l'ordinador sense haver de tenir un dispositiu físic sempre connectat o un emulador obert. No obstant això, haurem de provar la nostra aplicació cada cert temps a la plataforma Android per assegurar-nos que tot el que anem fent és compatible.

2.3 Desenvolupament del joc

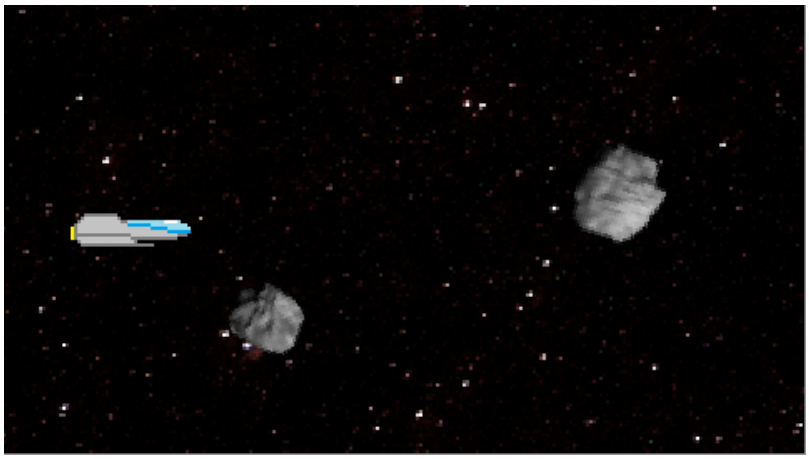
Una vegada tenim el projecte configurat i podent-se executar a l'ordinador, haurem de començar a desenvolupar el nostre joc. Començarem per les classes principals i afegirem els gràfics, els sons i la lògica de tots els components del joc.

Podeu descarregar els recursos de l'aplicació des dels annexos.

2.3.1 Classes del joc

El nostre joc estarà compost per diverses classes, cadascuna de les quals tindrà una funció específica. Cal, doncs, fer un repàs a cada classe del joc i anar-les implementant a poc a poc. A la figura 2.5 teniu una imatge del joc resultant que servirà d'ajuda per a la comprensió de les classes necessàries.

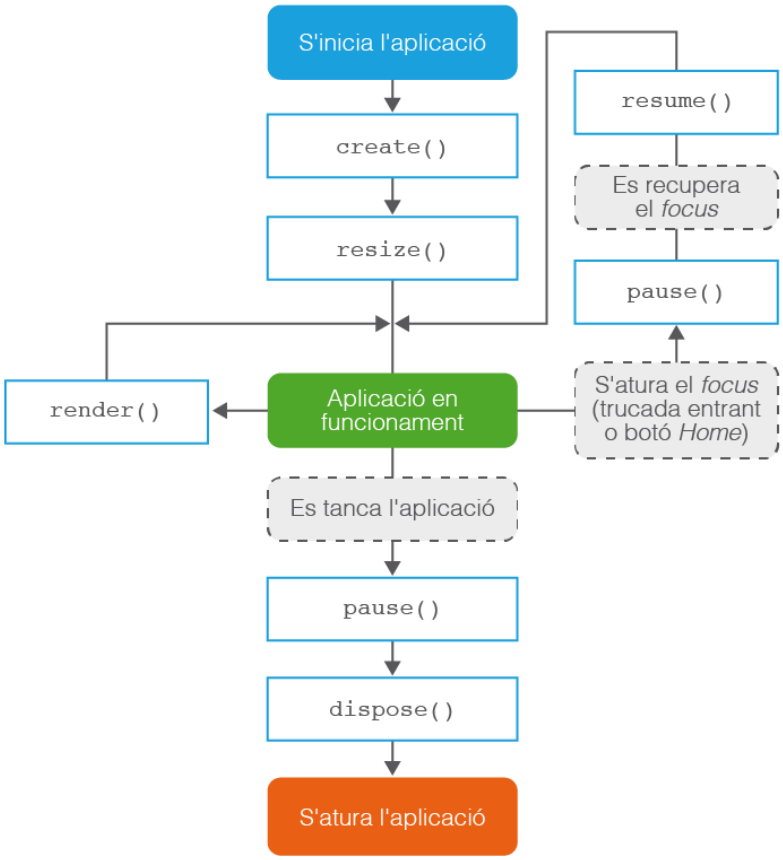
Figura 2.5. Joc resultant



- **SpaceRace** *(cat.xtec.ioc)*

Aquesta classe serà l'encarregada de carregar els recursos i controlar les pantalles del joc. Estendrà la classe `Game` i farem servir el mètode `setScreen(Screen screen)` per afegir la pantalla principal. Els `ApplicationListener`, que implementa la classe `Game`, tenen un cicle de vida similar al de les activitats d'Android com podem veure a la figura 2.6.

Figura 2.6. Cicle de vida d'ApplicationListener



Els mètodes disponibles són:

TAULA 2.1.

Mètode	Descripció
create()	Primer mètode que s'executa quan l'aplicació ha estat creada
resize(int width, int height)	Es crida una vegada després del mètode `create()` i cada vegada que es canvia la mida de l'aplicació i el joc no està a l'estat de pausa. Els paràmetres que rep són les noves dimensions de la pantalla
pause()	Quan l'aplicació perd el primer pla. S'executa també abans del mètode `dispose()`. Aquí haurem de guardar l'estat del joc
resume()	Quan l'aplicació torna a primer pla
render()	Mètode que es crida des del bucle principal de l'aplicació quan s'ha de representar la pantalla. Les actualitzacions de la lògica del joc s'habituen a fer en aquest mètode
dispose()	Darrer mètode que s'executa abans de tancar l'aplicació

- **GameScreen** *(cat.xtec.ioc.screens)*

Serà la pantalla principal del joc, encarregada de definir el gestor d'*events* d'entrada i d'actualitzar els components principals del joc: l'*stage* i els *actors*. Té un cicle de vida similar a la interfície `AplicationListener`, però a més afegeix els mètodes `show()` i `hide()`, quan una pantalla és afegida amb el mètode `setScreen()` i quan és canviada per una altra pantalla respectivament.

- **AssetManager** *(cat.xtec.ioc.helpers)*

En aquesta classe definirem tots els recursos bé siguin gràfics o sonors de la nostra aplicació.

- **InputHandler** *(cat.xtec.ioc.helpers)*

Gestió dels *events* de l'usuari, hereta d'`InputProcessor`. Podrem controlar els *events*: *keyDown*, *keyUp*, *keyTyped*, *touchDown*, *touchUp*, *touchDragged*, *mouseMoved* i *scrolled*, aquest dos últims exclusivament per les aplicacions d'escriptori i que fan referència al moviment del punter i de la roda de desplaçament respectivament.

- **Spacecraft** *(cat.xtec.ioc.objects)*

La nau del joc, serà un `Actor` del qual controlem la seva direcció segons els *events* rebuts per l'`InputHandler`.

- **Scrollable** *(cat.xtec.ioc.objects)*

Hereta de la classe `Actor`. Representa elements que disposen de desplaçament horitzontal al joc, en cada actualització els mourem segons la velocitat indicada. El fons de pantalla i els asteroides seran subclasses seva.

- **Asteroid** *(cat.xtec.ioc.objects)*

Classe que hereta d'`Scrollable`. Hi necessitarem controlar les possibles col·lisions amb la nau.

- **Background** *(cat.xtec.ioc.objects)*

Classe que hereta d'`Scrollable`, no té cap propietat ni mètode especial a banda de sobreescriure el mètode `draw`.

- **ScrollHandler** *(cat.xtec.ioc.objects)*

Classe encarregada de crear i actualitzar tots els elements que són `Scrollable`.

Ara que ja tenim una idea bàsica de les classes, cal veure quines són les implementacions per cada classe.

2.3.2 Implementació inicial

Començarem editant la classe `SpaceRace` que és l'única classe que tenim disponible en aquest moment. El primer que farem és modificar-ne la definició i fer que hereti de `Game` i no de `ApplicationAdapter`. Així, la definició de la classe quedarà de la següent manera:

```
1  public class SpaceRace extends Game {
```

Eliminem tot el contingut de `create()` i de `render()` així com les variables d'instància `batch` i `img`.

En aquesta classe iniciarem tots els recursos de l'aplicació i posarem la pantalla principal de l'aplicació. Abans de res, comprovarem el cicle de vida de l'aplicació amb el següent codi:

```
1  package cat.xtec.ioc;
2
3  import com.badlogic.gdx.Game;
4  import com.badlogic.gdx.Gdx;
5
6  public class SpaceRace extends Game {
7
8      @Override
9      public void create() {
10         Gdx.app.log("LifeCycle", "create()");
11     }
12
13     @Override
```

```
14      public void resize(int width, int height) {
15          super.resize(width, height);
16          Gdx.app.log("LifeCycle", "resize(" + Integer.toString(width) + ", " +
                Integer.toString(height) + ")");
17      }
18
19      @Override
20      public void pause() {
21          super.pause();
22          Gdx.app.log("LifeCycle", "pause()");
23      }
24
25      @Override
26      public void resume() {
27          super.resume();
28          Gdx.app.log("LifeCycle", "resume()");
29      }
30
31      @Override
32      public void render() {
33          super.render();
34          //Gdx.app.log("LifeCycle", "render()");
35      }
36
37      @Override
38      public void dispose() {
39          super.dispose();
40          Gdx.app.log("LifeCycle", "dispose()");
41
42      }
43  }
```

`Gdx.app.log(String tag, String msg)` és l'equivalent al `Log.d(String tag, String msg)` que estem acostumats a fer servir. Si executem l'aplicació i la tanquem obtindrem el següent resultat:

```
1  LifeCycle: create()
2  LifeCycle: resize(640, 480)
3  LifeCycle: pause()
4  LifeCycle: dispose()
```

Si descomenteu la línia `//Gdx.app.log(“LifeCycle”, “render()”);` veureu com aquest mètode es crida contínuament entre els mètodes `resize()` i `pause()`. Modifiqueu la classe per tal que tingui el següent codi:

```
1   package cat.xtec.ioc;
2
3   import com.badlogic.gdx.Game;
4
5   import cat.xtec.ioc.helpers.AssetManager;
6   import cat.xtec.ioc.screens.GameScreen;
7
8   public class SpaceRace extends Game {
9
10      @Override
11      public void create() {
12
13          // A l'iniciar el joc carreguem els recursos
14          AssetManager.load();
15          // I definim la pantalla principal com a la pantalla
16          setScreen(new GameScreen());
17
18      }
19
20      // Cridem per descartar els recursos carregats.
21      @Override
```

```
22      public void dispose() {
23          super.dispose();
24          AssetManager.dispose();
25      }
26  }
```

En aquest moment l'aplicació ens donarà error ja que no tenim disponibles les classes `AssetManager` ni `GameScreen`.

Crearem la classe `GameScreen` però abans, crearem el paquet `screens` que emmagatzemarà les pantalles del joc: fem clic al botó dret sobre el paquet *cat.xtec.ioc* i fem clic a *New/Package* i posem de nom **screens**. Fem ara clic en el nou paquet i creem una nova classe que anomenarem **GameScreen**, que serà la pantalla principal del joc. Aquesta classe implementarà la interfície `com.badlogic.gdx.Screen`, afegiu l'`implements Screen` a la definició de la classe i solucioneu l'error que ens diu l'Android Studio amb *Implement methods*, així afegirem els mètodes necessaris per implementar la classe *Screen*. La classe quedarà de la següent manera:

```
1   package cat.xtec.ioc.screens;
2
3   import com.badlogic.gdx.Screen;
4
5   public class GameScreen implements Screen {
6
7
8       @Override
9       public void show() {
10
11      }
12
13      @Override
14      public void render(float delta) {
15
16      }
17
18      @Override
19      public void resize(int width, int height) {
20
21      }
22
23      @Override
24      public void pause() {
25
26      }
27
28      @Override
29      public void resume() {
30
31      }
32
33      @Override
34      public void hide() {
35
36      }
37
38      @Override
39      public void dispose() {
40
41      }
42
43  }
```

Hem creat una de les dues classes que necessitàvem, creem el *package* `cat.xtec.ioc.helpers` i creem la classe `AssetManager` per solucionar el segon error.

Necessitem afegir dos mètodes `static` a la classe: `load()` i `dispose()` obtenint com a resultat la següent classe:

```
1  package cat.xtec.ioc.helpers;
2
3
4  public class AssetManager {
5
6      public static void load() {
7      }
8
9      public static void dispose() {
10     }
11
12 }
```

I amb això ja serem capaços d'executar la nostra aplicació, si afegim a més les línies:

```
1  Gdx.gl.glClearColor(1, 0, 0, 1);
2  Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

El mètode `render()` s'executarà contínuament, sent un bon candidat per ser el bucle principal de l'aplicació.

Al mètode `render(float delta)` de la classe `GameScreen`, tindrem com a resultat una pantalla amb fons vermell.

Deixarem de banda la classe `AssetManager` fins que tractem els gràfics i els sons de l'aplicació i ens centrarem en la classe `GameScreen`.

> Un **stage** és una escena 2D que conté `Actors` i que gestiona els *events* d'entrada i *viewports* (que veurem a la secció **Gràfics i sons**).
> Un **actor** és un node de l'`stage` que té propietats i sobre el qual es poden realitzar accions.

`GameScreen` és la classe que s'encarregarà de crear l'*stage* i els actors, és a dir, l'encarregada de controlar els elements principals del joc. La nostra classe haurà de tenir les següents variables d'instància:

```
1  private Stage stage;
2      private Spacecraft spacecraft;
3      private ScrollHandler scrollHandler;
```

Les classes `Spacecraft` i `ScrollHandler` les crearem més endavant. Els tres objectes els iniciarem al constructor de la pantalla:

```
1  public GameScreen() {
2
3          // Creem l'stage
4          stage = new Stage();
5
6          // Creem la nau i la resta d'objectes
7          spacecraft = new Spacecraft();
8          scrollHandler = new ScrollHandler();
9
```

```
10          // Afegim els actors a l'stage
11          stage.addActor(scrollHandler);
12          stage.addActor(spacecraft);
13
14      }
```

Més endavant necessitarem afegir paràmetres en la creació dels objectes però de moment ho deixarem així.

La classe `stage` disposa dels mètodes `draw()` i `act(float delta)` que cridarà als mètodes `draw()` i `act(float delta)` de tots els seus `actors`. Com que a la pantalla principal haurem de controlar l'actualització dels elements del joc, modificarem el mètode `render(float delta)` per tal que tingui el següent codi:

```
1  @Override
2      public void render(float delta) {
3
4          // Dibuixem i actualitzem tots els actors de l'stage
5          stage.draw();
6          stage.act(delta);
7
8      }
```

El valor **delta** marcarà la diferència en segons respecte l'anterior crida a `render`, fent servir aquest valor aconseguirem fer aplicacions amb moviments independents del *frame-rate*. Si fem la divisió **1/delta** obtindrem els *fps* (*frames per second)* de l'aplicació.

El valor `delta` el podem obtenir des de qualsevol lloc de l'aplicació amb `Gdx.graphics.getDeltaTime()`.

Hem de destacar el paràmetre `float delta`, que passarem al mètode `act` per tal que en actualitzar-se ho faci independentment del *frame-rate* i l'aplicació vagi a la mateixa "velocitat" independentment del dispositiu en què es faci servir.

Afegirem també a la classe els *getters* dels elements. Feu clic a *Code/-Generate.../Getter* i seleccioneu els tres elements, l'`stage`, l'`spacecraft` i l'`scrollhandler`, que donaran com a resultat el següent codi:

```
1  public Spacecraft getSpacecraft() {
2          return spacecraft;
3      }
4
5      public Stage getStage() {
6          return stage;
7      }
8
9      public ScrollHandler getScrollHandler() {
10         return scrollHandler;
11     }
```

Passarem ara a definir els elements del joc i després modificarem aquestes classes.

2.3.3 Elements del joc

Per tal de tenir organitzats els elements del joc, el primer que farem és crear el paquet cat.xtec.ioc.objects. Aquí haurem de crear les classes per a la nau espacial (`Spacecraft`), els asteroides (`Asteroid`) i el fons de pantalla (`Background`). Aquests dos últims tindran una classe pare comuna que anomenarem `Scrollable`.

Spacecraft

FIGURA 2.7. Spritesheet de la nau

Per a la nau espacial, necessitarem conèixer la posició que ocupa, les dimensions (alt i ample) i la direcció (si es troba "recta", pujant o baixant, tal com podem veure a la figura 2.7).

La nostra nau espacial serà un `Actor` de l'`stage`. Crearem la classe `Spacecraft` i modificarem la definició de la classe per tal que hereti d'`actor`:

```
1  public class Spacecraft extends Actor {
```

Afegirem a la classe les variables d'instància corresponents i crearem el constructor. Crearem a més tres constants de classe per controlar la direcció. La classe quedarà de la següent manera:

```
1  package cat.xtec.ioc.objects;
2
3  import com.badlogic.gdx.math.Vector2;
4  import com.badlogic.gdx.scenes.scene2d.Actor;
5
6  public class Spacecraft extends Actor {
7
8      // Diferents posicions de l'Spacecraft: recta, pujant i baixant
9      public static final int SPACECRAFT_STRAIGHT = 0;
10     public static final int SPACECRAFT_UP = 1;
11     public static final int SPACECRAFT_DOWN = 2;
12
13         // Paràmetres de l'Spacecraft
14     private Vector2 position;
15     private int width, height;
16     private int direction;
17
18     public Spacecraft(float x, float y, int width, int height) {
19
20         // Inicialitzem els arguments segons la crida del constructor
21         this.width = width;
22         this.height = height;
23         position = new Vector2(x, y);
24
25         // Inicialitzem l'Spacecraft a l'estat normal
26         direction = SPACECRAFT_STRAIGHT;
27
```

```
28      }
29  }
```

A la wiki oficial de LibGDX podeu trobar informació sobre la classe Vector2D

Afegim els *getters* de les propietats i afegim els mètodes `goUp()`, `goDown()` i `goStraight()` que canviaran la direcció depenent de si la nau puja, baixa o va recta.

I ens quedarà el mètode més important, el mètode `act(float delta)` que es cridarà quan la nau estigui com a `actor` en un `stage` i aquest faci una crida al seu mètode `act`. Aquest mètode haurà d'actualitzar la nau canviant la seva posició segons la direcció i la velocitat.

El nostre joc tindrà un sistema de coordenades *Y-Down*, que vol dir que la coordenada 0 de la Y serà a la part superior de la pantalla, una Y positiva anirà en direcció descendent i una Y negativa serà ascendent. Per moure la nau, ens trobarem en dos casos:

- Nau ascendent: la posició de la Y l'haurem de disminuir multiplicant la velocitat pel valor `delta` (per aconseguir un moviment independent del *frame-rate*), i ho farem sempre que la nova posició no sigui menor a 0.
- Nau descendent: la posició de la Y l'haurem d'incrementar multiplicant la velocitat pel valor `delta`, i ho farem sempre que la nova posició més l'alçada de la nau no sigui major que l'alçada de la pantalla.

Per evitar que la nau surti de la pantalla mentre realitza un moviment descendent necessitem conèixer-ne l'alçada. Com que serà un valor que consultarem força sovint i que volem tenir disponible per canviar d'una manera senzilla, crearem una classe que anomenarem *Settings* en la qual emmagatzemarem constants que ens seran útils des de distintes classes. Crearem un nou paquet Java de nom **cat.xtec.ioc.utils** i una classe **Settings** que contindrà el següent codi:

```
1  package cat.xtec.ioc.utils;
2
3  public class Settings {
4
5      // Mida del joc, s'escalarà segons la necessitat
6      public static final int GAME_WIDTH = 240;
7      public static final int GAME_HEIGHT = 135;
8
9      // Propietats de la nau
10     public static final float SPACECRAFT_VELOCITY = 50;
11     public static final int SPACECRAFT_WIDTH = 36;
12     public static final int SPACECRAFT_HEIGHT = 15;
13     public static final float SPACECRAFT_STARTX = 20;
14     public static final float SPACECRAFT_STARTY = GAME_HEIGHT/2 -
           SPACECRAFT_HEIGHT/2;
15
16 }
```

Hem afegit també les constants que serviran per definir els paràmetres de la nau: velocitat, amplada, alçada i punt inicial. A 20 píxels de la vora esquerra i centrada en l'eix vertical (alçada del joc menys la meitat de l'alçada de la nau).

El mètode `act(float delta)` es cridarà sempre que es faci una crida a `stage.act(delta)`.

El codi del mètode encarregat d'actualitzar la nau podria ser el següent:

```
public void act(float delta) {

        // Movem l'Spacecraft depenent de la direcció controlant que no surti
            de la pantalla
        switch (direction) {
            case SPACECRAFT_UP:
                if (this.position.y - Settings.SPACECRAFT_VELOCITY * delta >=
                    0) {
                    this.position.y -= Settings.SPACECRAFT_VELOCITY * delta;
                }
                break;
            case SPACECRAFT_DOWN:
                if (this.position.y + height + Settings.SPACECRAFT_VELOCITY *
                    delta <= Settings.GAME_HEIGHT) {
                    this.position.y += Settings.SPACECRAFT_VELOCITY * delta;
                }
                break;
            case SPACECRAFT_STRAIGHT:
                break;
        }
    }
```

Si el moviment és cap a dalt, controlem que la coordenada Y, si li apliquem l'increment de posició (restant la velocitat multiplicada pel valor `delta`), no sigui mai menor de 0. En el cas de moviment descendent, hem de controlar que la Y més l'alçada de la nau i sumant-li el nou increment de posició (velocitat per `delta`) no superi els límits de la pantalla, és a dir, el `GAME_HEIGHT`.

El codi complet d'`Spacecraft` serà el següent:

```
package cat.xtec.ioc.objects;

import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.scenes.scene2d.Actor;

import cat.xtec.ioc.utils.Settings;

public class Spacecraft extends Actor {

    // Distintes posicions de l'Spacecraft: recta, pujant i baixant
    public static final int SPACECRAFT_STRAIGHT = 0;
    public static final int SPACECRAFT_UP = 1;
    public static final int SPACECRAFT_DOWN = 2;

    // Paràmetres de l'Spacecraft
    private Vector2 position;
    private int width, height;
    private int direction;

    public Spacecraft(float x, float y, int width, int height) {

        // Inicialitzem els arguments segons la crida del constructor
        this.width = width;
        this.height = height;
        position = new Vector2(x, y);

        // Inicialitzem l'Spacecraft a l'estat normal
        direction = SPACECRAFT_STRAIGHT;

    }

    public void act(float delta) {

        // Movem l'Spacecraft depenent de la direcció controlant que no surti
            de la pantalla
        switch (direction) {
```

```
36              case SPACECRAFT_UP:
37                  if (this.position.y - Settings.SPACECRAFT_VELOCITY * delta >=
                        0) {
38                      this.position.y -= Settings.SPACECRAFT_VELOCITY * delta;
39                  }
40                  break;
41              case SPACECRAFT_DOWN:
42                  if (this.position.y + height + Settings.SPACECRAFT_VELOCITY *
                        delta <= Settings.GAME_HEIGHT) {
43                      this.position.y += Settings.SPACECRAFT_VELOCITY * delta;
44                  }
45                  break;
46              case SPACECRAFT_STRAIGHT:
47                  break;
48          }
49      }
50
51      // Getters dels atributs principals
52      public float getX() {
53          return position.x;
54      }
55
56      public float getY() {
57          return position.y;
58      }
59
60      public float getWidth() {
61          return width;
62      }
63
64      public float getHeight() {
65          return height;
66      }
67
68      // Canviem la direcció de l'Spacecraft: Puja
69      public void goUp() {
70          direction = SPACECRAFT_UP;
71      }
72
73      // Canviem la direcció de l'Spacecraft: Baixa
74      public void goDown() {
75          direction = SPACECRAFT_DOWN;
76      }
77
78      // Posem l'Spacecraft al seu estat original
79      public void goStraight() {
80          direction = SPACECRAFT_STRAIGHT;
81      }
82
83  }
```

A la classe `GameScreen` havíem creat un nou objecte `Spacecraft` sense passar-li cap paràmetre i el constructor que hem definit necessita quatre paràmetres: coordenada X, coordenada Y, amplada i alçada. Tornem a la classe `GameScreen` i modifiquem del constructor la línia `spacecraft = new Spacecraft()` per:

```
1  spacecraft = new Spacecraft(Settings.SPACECRAFT_STARTX, Settings.
       SPACECRAFT_STARTY, Settings.SPACECRAFT_WIDTH, Settings.SPACECRAFT_HEIGHT);
```

Scrollables

Crearem, ara, els objectes que ens falten: els asteroides i el fons de pantalla. Tots dos tenen coses en comú: són descendents d'`actor`, tenen un desplaçament lateral

i es van reutilitzant cada vegada que desapareixen pel costat esquerre (valor de X=0). Crearem al paquet `cat.xtec.ioc.objects` una classe pare anomenada **scrollable** i dos descendents d'aquesta: **Asteroid** i **Background**.

La classe `scrollable` heretarà d'`actor` i necessitarem:

- `Vector2 position`: vector de posició.
- `float velocity`: `float` amb la velocitat de l'element. Serà variable segons l'element.
- `float width, height`: amplada i alçada dels elements.
- `boolean leftOfScreen`: per saber si l'element està fora de la pantalla, quan sobrepassa el 0 de l'eix d'X.

Creem la classe amb els atributs, el constructor, els *getters*, la funció `getTailX()` que retornarà la posició de l'element més la seva amplada i la funció `reset (float newX)` que assignarà una nova posició a l'element i posarà a `false` el booleà que indica que ha sortit de la pantalla.

La classe `scrollable` quedarà així:

```
1  package cat.xtec.ioc.objects;
2
3  import com.badlogic.gdx.math.Vector2;
4  import com.badlogic.gdx.scenes.scene2d.Actor;
5
6  public class Scrollable extends Actor {
7
8      protected Vector2 position;
9      protected float velocity;
10     protected float width;
11     protected float height;
12     protected boolean leftOfScreen;
13
14     public Scrollable(float x, float y, float width, float height, float
           velocity) {
15         position = new Vector2(x, y);
16         this.velocity = velocity;
17         this.width = width;
18         this.height = height;
19         leftOfScreen = false;
20
21     }
22
23     public void reset(float newX) {
24         position.x = newX;
25         leftOfScreen = false;
26     }
27
28     public boolean isLeftOfScreen() {
29         return leftOfScreen;
30     }
31
32     public float getTailX() {
33         return position.x + width;
34     }
35
36     public float getX() {
37         return position.x;
38     }
39
```

```
    public float getY() {
        return position.y;
    }

    public float getWidth() {
        return width;
    }

    public float getHeight() {
        return height;
    }


}
```

Faltarà implementar el mètode principal: `act(float delta)`. Aquest mètode s'encarregarà de desplaçar l'element en l'eix d'X. Controlarem a més el booleà `leftOfScreen`, que posarem a `true` quan la posició en l'eix X més l'amplada de l'objecte sigui menor de 0.

Així, el mètode quedarà de la següent manera:

```
public void act(float delta) {

        // Desplacem l'objecte en l'eix d'X
        position.x += velocity * delta;

        // Si es troba fora de la pantalla canviem la variable a true
        if (position.x + width < 0) {
            leftOfScreen = true;
        }
    }
```

Les dues classes filles, `asteroid` i `background`, de moment les deixarem amb la implementació mínima, quedant de la següent manera:

Classe Asteroid:

```
package cat.xtec.ioc.objects;

public class Asteroid extends Scrollable {
    public Asteroid(float x, float y, float width, float height, float velocity
        ) {
        super(x, y, width, height, velocity);
    }
}
```

Classe Background:

```
package cat.xtec.ioc.objects;

public class Background extends Scrollable {
    public Background(float x, float y, float width, float height, float
        velocity) {
        super(x, y, width, height, velocity);
    }
}
```

Ens quedarà per últim definir la classe **scrollHandler** que servirà per controlar tots els objectes del tipus `scrollable`. En aquesta classe necessitarem un mètode que ens retornarà un `float` aleatori entre dos valors. Crearem una classe on guardarem

mètodes que puguin ser útils des de qualsevol classe del joc. Creeu al paquet `cat.xtec.ioc.utils` la classe `methods` i que tindrà com a contingut el següent:

```
package cat.xtec.ioc.utils;

import java.util.Random;

public class Methods {

    // Mètode que torna un float aleatori entre un mínim i un màxim
    public static float randomFloat(float min, float max) {
        Random r = new Random();
        return r.nextFloat() * (max - min) + min;

    }
}
```

Haurem d'afegir també les següents constants a la classe `Settings`:

```
// Rang de valors per canviar la mida de l'asteroide
    public static final float MAX_ASTEROID = 1.5f;
    public static final float MIN_ASTEROID = 0.5f;

    // Configuració scrollable
    public static final int ASTEROID_SPEED = -150;
    public static final int ASTEROID_GAP = 75;
    public static final int BG_SPEED = -100;
```

La classe `scrollHandler` crearà els `asteroid` (que s'aniran reutilitzant) i dos `background` i serà l'encarregada d'actualitzar-los. Aquesta classe heretarà de `group` (`com.badlogic.gdx.scenes.scene2d.Group`), que és un conjunt d'`actors`.

El codi inicial de la classe serà el següent:

```
package cat.xtec.ioc.objects;

import com.badlogic.gdx.scenes.scene2d.Group;

import java.util.ArrayList;
import java.util.Random;

public class ScrollHandler extends Group {

    // Fons de pantalla
    Background bg, bg_back;

    // Asteroides
    int numAsteroids;
    ArrayList<Asteroid> asteroids;

    // Objecte random
    Random r;

    public ScrollHandler() {
    }
}
```

Un **group** és un node d'una escena 2D que conté actors. Els actors es representaran segons l'ordre d'inserció al grup i igual que aquests, disposen d'un mètode `act` i un mètode `draw`.

Començarem creant els dos fons que s'aniran concatenant per donar aquesta sensació de fons infinit. El seu codi, que afegirem al constructor d'`scrollHandler`, serà el següent:

```
//Creem els dos fons
        bg = new Background(0, 0, Settings.GAME_WIDTH * 2, Settings.GAME_HEIGHT
            , Settings.BG_SPEED);
        bg_back = new Background(bg.getTailX(), 0, Settings.GAME_WIDTH * 2,
            Settings.GAME_HEIGHT, Settings.BG_SPEED);

        //Afegim els fons (actors) al grup
        addActor(bg);
        addActor(bg_back);
```

El primer fons començarà en la coordenada (0,0) i tindrà una amplada el doble que la mida de la pantalla i la mateixa alçada. També li passarem la velocitat en què es mourà. Pel que fa al segons fons, començarà a la cua del fons principal i amb la Y=0, les dimensions i la velocitat seran les mateixes. Una vegada creats els afegim al `group` amb el mètode `addActor(Actor actor)`.

Pels que fa als `asteroid`, tindran una amplada i alçada aleatòria que serà entre 0.5 i 1.5 vegades la seva mida original (en aquest cas 34 píxels). Així, guardarem a un `float` la nova mida i la farem servir per calcular les propietats. Necessitarem calcular les coordenades X i Y segons la nova mida:

- Coordenada X: en el cas del primer `asteroid` serà l'ample del joc i pels altres la posició de la cua (`getTailX()`) de l'anterior asteroide més una separació que definirem a `settings` (`ASTEROID_GAP`).
- Coordenada Y: serà un valor aleatori entre 0 i l'alçada total de la pantalla menys l'alçada de l'asteroide que hem calculat.

Crearem un `private ArrayList<Asteroid>` anomenat **asteroids** que contindrà els `asteroid` que anirem generant. Inicialment crearem 3 asteroides però ho deixarem preparat per tal que afegir-ne de nous no sigui complex. El codi següent l'afegirem després de la creació dels fons:

```
// Creem l'objecte random
        r = new Random();

        // Comencem amb 3 asteroides
        numAsteroids = 3;

        // Creem l'ArrayList
        asteroids = new ArrayList<Asteroid>();

        // Definim una mida aleatòria entre el mínim i el màxim
        float newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
            MAX_ASTEROID) * 34;

        // Afegim el primer asteroide a l'array i al grup
        Asteroid asteroid = new Asteroid(Settings.GAME_WIDTH, r.nextInt(
            Settings.GAME_HEIGHT - (int) newSize), newSize, newSize, Settings.
            ASTEROID_SPEED);
        asteroids.add(asteroid);
        addActor(asteroid);

        // Des del segon fins l'últim asteroide
        for (int i = 1; i < numAsteroids; i++) {
```

```
            // Creem la mida aleatòria
            newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
                MAX_ASTEROID) * 34;
            // Afegim l'asteroide
            asteroid = new Asteroid(asteroids.get(asteroids.size() - 1).
                getTailX() + Settings.ASTEROID_GAP, r.nextInt(Settings.
                GAME_HEIGHT - (int) newSize), newSize, newSize, Settings.
                ASTEROID_SPEED);
            // Afegim l'asteroide a l'ArrayList
            asteroids.add(asteroid);
            // Afegim l'asteroide al grup d'actors
            addActor(asteroid);
        }
```

Cadascun dels asteroides s'haurà d'afegir a l'`ArrayList` i també al `group`.

Cada `group` té un mètode `act(float delta)` comú per a tots els `actor`, al mètode `act` de l'`scrollHandler`. Comprovarem si algun dels seus `actors` ha deixat de ser visible per la pantalla i en cas de ser així, el reiniciarem assignat-li una nova posició: en el cas del fons li assignarem la posició de la cua de l'altre fons i en el cas dels asteroides agafarem la cua de l'anterior asteroide (en el cas de que sigui el primer asteroide agafarem la cua del darrer).

Per generar el codi del mètode `act` feu clic a *Code/Override Methods...* i escolliu el mètode `act(delta:float):void`. Una vegada generat, afegiu el següent codi per fer les comprovacions dels objectes:

```
@Override
    public void act(float delta) {
        super.act(delta);
        // Si algun element es troba fora de la pantalla, fem un reset de l'
            element
        if (bg.isLeftOfScreen()) {
            bg.reset(bg_back.getTailX());

        } else if (bg_back.isLeftOfScreen()) {
            bg_back.reset(bg.getTailX());

        }

        for (int i = 0; i < asteroids.size(); i++) {

            Asteroid asteroid = asteroids.get(i);
            if (asteroid.isLeftOfScreen()) {
                if (i == 0) {
                    asteroid.reset(asteroids.get(asteroids.size() - 1).getTailX
                        () + Settings.ASTEROID_GAP);
                } else {
                    asteroid.reset(asteroids.get(i - 1).getTailX() + Settings.
                        ASTEROID_GAP);
                }
            }
        }
    }
```

i finalment afegirem el *getter* per obtenir l'`ArrayList`, quedant la classe de la següent manera:

```
package cat.xtec.ioc.objects;

import com.badlogic.gdx.scenes.scene2d.Group;

import java.util.ArrayList;
```

```
6  import java.util.Random;
7
8  import cat.xtec.ioc.utils.Methods;
9  import cat.xtec.ioc.utils.Settings;
10
11 public class ScrollHandler extends Group {
12
13     // Fons de pantalla
14     Background bg, bg_back;
15
16     // Asteroides
17     int numAsteroids;
18     ArrayList<Asteroid> asteroids;
19
20     // Objecte Random
21     Random r;
22
23     public ScrollHandler() {
24
25         // Creem els dos fons
26         bg = new Background(0, 0, Settings.GAME_WIDTH * 2, Settings.GAME_HEIGHT
               , Settings.BG_SPEED);
27         bg_back = new Background(bg.getTailX(), 0, Settings.GAME_WIDTH * 2,
               Settings.GAME_HEIGHT, Settings.BG_SPEED);
28
29         // Afegim els fons al grup
30         addActor(bg);
31         addActor(bg_back);
32
33         // Creem l'objecte random
34         r = new Random();
35
36         // Comencem amb 3 asteroides
37         numAsteroids = 3;
38
39         // Creem l'ArrayList
40         asteroids = new ArrayList<Asteroid>();
41
42         // Definim una mida aleatòria entre el mínim i el màxim
43         float newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
               MAX_ASTEROID) * 34;
44
45         // Afegim el primer asteroide a l'array i al grup
46         Asteroid asteroid = new Asteroid(Settings.GAME_WIDTH, r.nextInt(
               Settings.GAME_HEIGHT - (int) newSize), newSize, newSize, Settings.
               ASTEROID_SPEED);
47         asteroids.add(asteroid);
48         addActor(asteroid);
49
50         // Des del segon fins l'últim asteroide
51         for (int i = 1; i < numAsteroids; i++) {
52             // Creem la mida aleatòria
53             newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
                   MAX_ASTEROID) * 34;
54             // Afegim l'asteroide
55             asteroid = new Asteroid(asteroids.get(asteroids.size() - 1).
                   getTailX() + Settings.ASTEROID_GAP, r.nextInt(Settings.
                   GAME_HEIGHT - (int) newSize), newSize, newSize, Settings.
                   ASTEROID_SPEED);
56             // Afegim l'asteroide a l'ArrayList
57             asteroids.add(asteroid);
58             // Afegim l'asteroide al grup d'actors
59             addActor(asteroid);
60         }
61
62     }
63
64     @Override
65     public void act(float delta) {
66         super.act(delta);
```

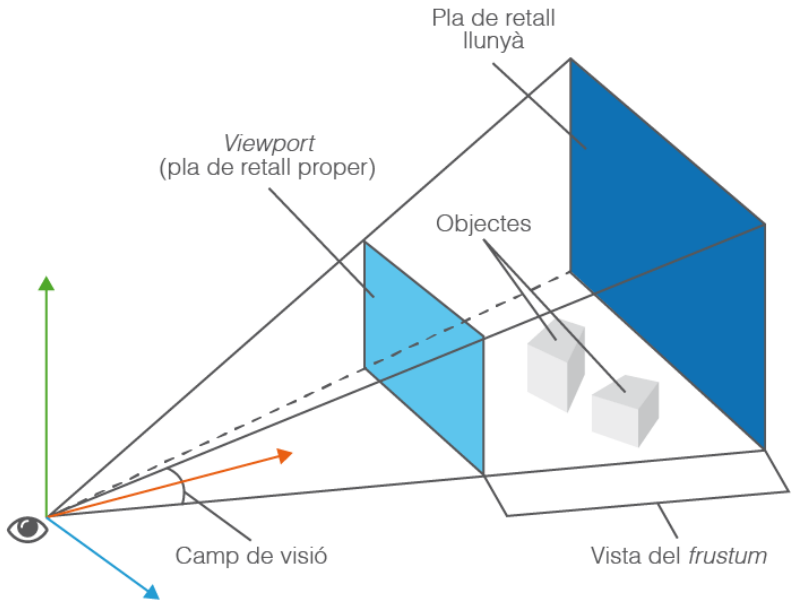
```
67          // Si algun element es troba fora de la pantalla, fem un reset de l'
                element
68          if (bg.isLeftOfScreen()) {
69              bg.reset(bg_back.getTailX());
70
71          } else if (bg_back.isLeftOfScreen()) {
72              bg_back.reset(bg.getTailX());
73
74          }
75
76          for (int i = 0; i < asteroids.size(); i++) {
77
78              Asteroid asteroid = asteroids.get(i);
79              if (asteroid.isLeftOfScreen()) {
80                  if (i == 0) {
81                      asteroid.reset(asteroids.get(asteroids.size() - 1).getTailX
                            () + Settings.ASTEROID_GAP);
82                  } else {
83                      asteroid.reset(asteroids.get(i - 1).getTailX() + Settings.
                            ASTEROID_GAP);
84                  }
85              }
86          }
87      }
88
89      public ArrayList<Asteroid> getAsteroids() {
90          return asteroids;
91      }
92  }
```

En aquest moment tenim tots els objectes preparats i si executem el projecte s'estaran actualitzant els valors tot i que no serem capaços de visualitzar-ho ja que no tenen cap representació gràfica.

2.3.4 Gràfics i sons

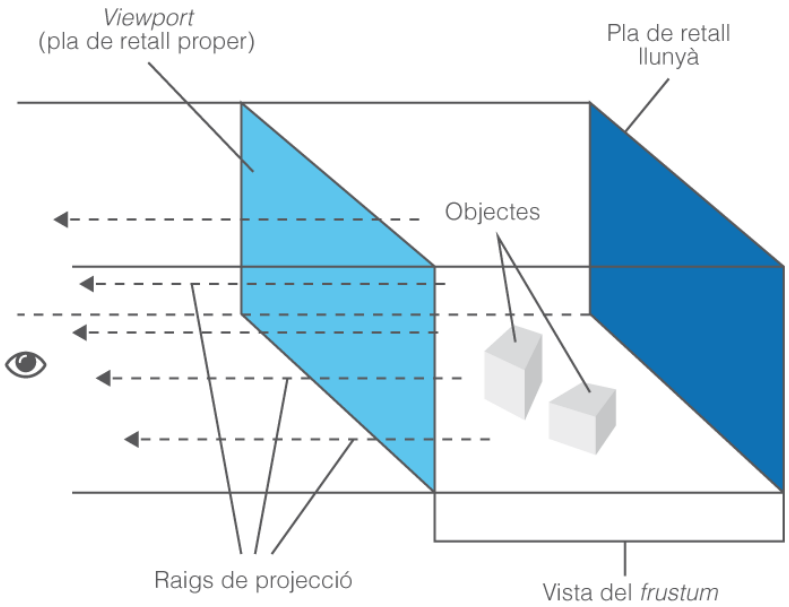
Totes les aplicacions que fan servir gràfics necessiten treballar amb la pantalla d'alguna manera, necessiten tenir definida una *camera* i un *viewport*. El *viewport* és una regió rectangular de la pantalla on es projecta una escena en 3D, és a dir, representa els objectes en 3D en un pla en 2D.

Figura 2.8. Escena 3D amb una projecció perspectiva



La ***OrthographicCamera*** és un tipus de càmera (determina els límits d'allò que pot veure el jugador) que fem servir en 2D per contra de la *PerspectiveCamera* que faríem servir en entorns 3D (on les dimensions i la profunditat dels objectes dependrà del camp de visió del jugador, tal i com veiem a la figura figura 2.8). En aquest tipus de càmera s'ignora la projecció dels objectes respecte el *viewport*, mantenint així la seva mida original (figura 2.9).

Figura 2.9. Projecció ortogonal



Prepararem, ara, els gràfics per tenir una representació visual del que està passant. Abans d'assignar els *sprites* a cada objecte, dibuixarem per pantalla les figures

geomètriques dels asteroides i de la nau per veure el resultat del que tenim fins ara. Ho podem fer de dues maneres: des de la classe `GameScreen` fent servir el mètode `render` o implementant els mètodes `draw()` de cadascun dels objectes `actor`. Per simplificar aquesta part ho farem des de la classe `GameScreen`. Aquesta, però, no serà la manera correcta, ja que s'hauria de fer al mètode `draw` de cada objecte, tal com farem per dibuixar els *sprites*.

Figures geomètriques

Per a la representació de figures geomètriques necessitarem un objecte `ShapeRenderer` i assignar-li una `OrthographicCamera` i un `viewport` a l'`stage`.

El `ShapeRenderer` serà l'encarregat de dibuixar per pantalla les figures geomètriques (rectangles i cercles) a la pantalla.

Obrim la classe `GameScreen` i definim les variables d'instància:

```
1 // Representació de figures geomètriques
2     private ShapeRenderer shapeRenderer;
3     // Per obtenir el batch de l'stage
4     private Batch batch;
```

L'objecte `batch` és l'element que ens permet dibuixar *sprites* a la pantalla. És un element molt 'pesat' del qual es recomana tenir una única instància. Quan creem un `stage` aquest ja disposa d'un `batch` que podrem recollir amb el mètode `stage.getBatch()` evitant-nos crear-ne un de nou cada cop que volem dibuixar *sprites* per pantalla.

Al constructor crearem la `OrthographicCamera` i el `viewport` per tal d'assignar-los a l'`stage`. Iniciarem el `ShapeRenderer` i recollirem l'objecte `batch` de l'`stage`:

- Inicialitzem el `ShapeRenderer`:

```
1 // Creem el ShapeRenderer
2         shapeRenderer = new ShapeRenderer();
```

El `FitViewport` respecta la relació d'aspecte entre les dimensions horitzontals i verticals a l'ampliar o reduir una aplicació, mentre que l'`StretchViewport` es redimensiona fins ocupar tota la pantalla, cosa que pot produir una deformació dels objectes ja que no manté la relació d'aspecte.

- Creem la càmera amb les dimensions del joc i la configurem perquè treballi amb el sistema de coordenades *Y-Down*:

```
1 // Creem la càmera de les dimensions del joc
2         OrthographicCamera camera = new OrthographicCamera(Settings.GAME_WIDTH,
              Settings.GAME_HEIGHT);
3         // Posant el paràmetre a true configurem la càmera perquè
4         // faci servir el sistema de coordenades Y_Down
5         camera.setToOrtho(true);
```

- Creem el *viewport*, en aquest cas, ens permetrà veure tot el que mostra la càmera (té les mateixes dimensions):

```
1 // Creem el viewport amb les mateixes dimensions que la càmera
2         StretchViewport viewport = new StretchViewport(Settings.GAME_WIDTH,
              Settings.GAME_HEIGHT , camera);
```

- Assignem el *Viewport* a l'`stage`:

```
// Creem l'stage i assignem el viewport
        stage = new Stage(viewport);
```

- Recuperem el `batch` per fer-lo servir posteriorment:

```
batch = stage.getBatch();
```

Amb aquests canvis, el constructor de `GameScreen` quedarà de la següent manera:

```
public GameScreen() {

        // Creem el ShapeRenderer
        shapeRenderer = new ShapeRenderer();

        // Creem la càmera de les dimensions del joc
        OrthographicCamera camera = new OrthographicCamera(Settings.GAME_WIDTH,
             Settings.GAME_HEIGHT);
        // Posant el paràmetre a true configurem la càmera perquè
        // faci servir el sistema de coordenades Y-Down
        camera.setToOrtho(true);

        // Creem el viewport amb les mateixes dimensions que la càmera
        StretchViewport viewport = new StretchViewport(Settings.GAME_WIDTH,
            Settings.GAME_HEIGHT , camera);

        // Creem l'stage i assignem el viewport
        stage = new Stage(viewport);

        batch = stage.getBatch();

        // Creem la nau i la resta d'objectes
        spacecraft = new Spacecraft(Settings.SPACECRAFT_STARTX, Settings.
            SPACECRAFT_STARTY, Settings.SPACECRAFT_WIDTH, Settings.
            SPACECRAFT_HEIGHT);
        scrollHandler = new ScrollHandler();

        // Afegim els actors a l'stage
        stage.addActor(scrollHandler);
        stage.addActor(spacecraft);


    }
```

Crearem un mètode `drawElements()` en el qual dibuixarem els elements. El primer que haurem de fer és configurar el `shapeRenderer` perquè dibuixi les figures geomètriques amb les mateixes propietats que el *viewport*, és a dir, amb les mateixes dimensions i fent servir un sistema de coordenades *Y-Down*. Per configurar-lo cridarem al mètode `shapeRenderer.setProjectionMatrix(batch.getProjectionMatrix());` que assignarà la matriu de projecció de l'`stage` al `ShapeRenderer`:

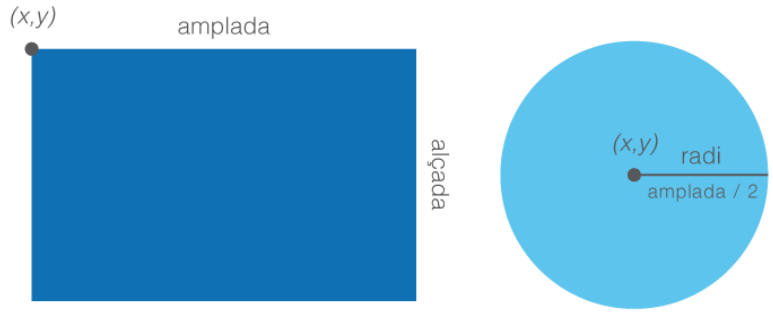
```
//Recollim les propietats del batch de l'stage
        shapeRenderer.setProjectionMatrix(batch.getProjectionMatrix());
```

Tot el que dibuixarem haurà d'anar entre els mètodes `shaperenderer.begin(ShapeType type)` i `shaperenderer.end()`. El `type` serà com volem dibuixar les figures geomètriques: `ShapeRenderer.ShapeType.Filled` per dibuixar l'interior o

`ShapeRenderer.ShapeType.Line` per dibuixar el contorn. Per configurar el color cridarem a `shaperenderer.color(Color color)`.

Per pintar els elements necessitarem un rectangle i diversos cercles, un per cada asteroide. Per dibuixar rectangles cridarem al mètode `rect(float x, float y, float width, float height)` i pels cercles el mètode `circle(float x, float y, float radius)` de l'objecte `ShapeRenderer`.

FIGURA 2.10. Propietats del rectangle i el cercle



Per pintar els rectangles necessitarem la posició de la cantonada superior esquerra i les mides d'ample i alt del rectangle, en aquest cas els valors X i Y de la nau, així com el seu ample i el seu alt; pels cercles necessitarem el punt central del cercle i la mida del radi. Aquests valors els podrem obtenir a partir de les característiques dels elements, tal com podem veure en la figura 2.10.

Els passos a seguir seran:

1. Per evitar l'efecte *flickering* necessitem esborrar la pantalla cada cop que dibuixem les figures geomètriques, ho farem pintant de negre la pantalla.
2. Configurem el `ShapeRenderer` i cridem al mètode `begin()`.
3. Definim el color i pintem la nau.
4. Recorrem tots els `asteroid` i pintem els tres primers de vermell, blau i groc, tots els que podem afegir després seran pintats de blanc.
5. Fem la crida a `end()` i s'enviarà tot el contingut per ser pintat.

El codi de `drawElements()` podria quedar de la següent manera:

```
1  private void drawElements(){
2
3          /* 1 */
4          // Pintem el fons de negre per evitar el "flickering"
5          Gdx.gl20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
6          Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
7
8          /* 2 */
9          // Recollim les propietats del Batch de l'Stage
10         shapeRenderer.setProjectionMatrix(batch.getProjectionMatrix());
11         // Inicialitzem el shaperenderer
12         shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);
13
```

```
        /* 3 */
        // Definim el color (verd)
        shapeRenderer.setColor(new Color(0, 1, 0, 1));
        // Pintem la nau
        shapeRenderer.rect(spacecraft.getX(), spacecraft.getY(), spacecraft.
            getWidth(), spacecraft.getHeight());

        /* 4 */
        // Recollim tots els Asteroid
        ArrayList<Asteroid> asteroids = scrollHandler.getAsteroids();
        Asteroid asteroid;

        for (int i = 0; i < asteroids.size(); i++) {

            asteroid = asteroids.get(i);
            switch (i) {
                case 0:
                    shapeRenderer.setColor(1,0,0,1);
                    break;
                case 1:
                    shapeRenderer.setColor(0,0,1,1);
                    break;
                case 2:
                    shapeRenderer.setColor(1,1,0,1);
                    break;
                default:
                    shapeRenderer.setColor(1,1,1,1);
                    break;
            }
            shapeRenderer.circle(asteroid.getX() + asteroid.getWidth()/2,
                asteroid.getY() + asteroid.getWidth()/2, asteroid.getWidth()
                /2);
        }
        /* 5 */
        shapeRenderer.end();
    }
```

Haurem de cridar a `drawElements()` des del mètode `render(float delta)`, afegiu la línia `drawElements();` al final del mètode perquè quedi de la següent manera:

```
@Override
    public void render(float delta) {

        // Dibuixem i actualitzem tots els actors de l'stage
        stage.draw();
        stage.act(delta);

        drawElements();

    }
```

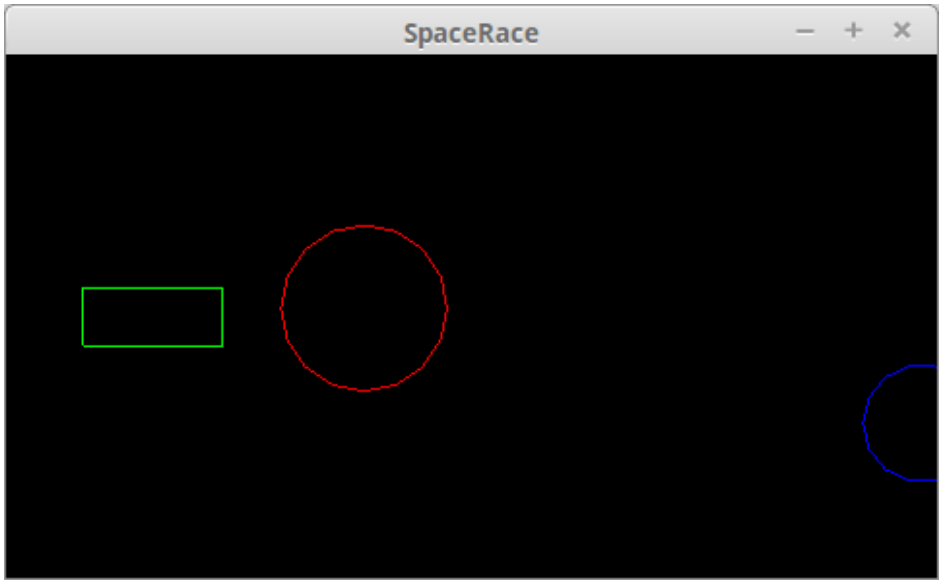
Abans d'executar el projecte i veure el resultat, cal configurar com s'executarà aquest treball com a aplicació d'escriptori. Obriu el fitxer *DesktopLauncher* del subprojecte *desktop* i modifiqueu el mètode `main`. Cal que definim una amplada i alçada de finestra dues vegades superior que com s'ha definit a `Settings`:

```
public static void main (String[] arg) {
    LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
    config.title = "SpaceRace";
    config.width = Settings.GAME_WIDTH * 2;
    config.height = Settings.GAME_HEIGHT * 2;
    new LwjglApplication(new SpaceRace(), config);
  }
```

Podeu canviar el gruix del `SapeType.Line` amb la propietat `Gdx.gl20.glLineWidth(int width)`.

Executeu el projecte i observeu el resultat, podeu veure la nau i els asteroides! Proveu de canviar la línia `shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);` per `shapeRenderer.begin(ShapeRenderer.ShapeType.Line);`. El resultat serà similar a la figura 2.11.

FIGURA 2.11. ShapeRenderer dibuixant els objectes



La nostra aplicació funciona correctament però si ens hi fixem amb més detall veurem que la posició dels asteroides no canvia quan reiniciem. Haurem d'afegir el mètode `reset(float newX)` a la classe `asteroid` amb el següent contingut:

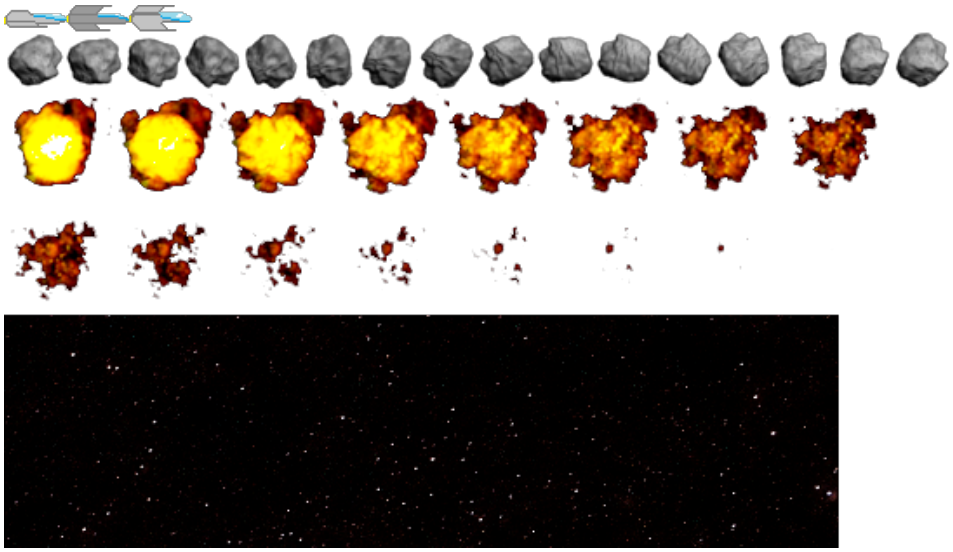
```
1  public void reset(float newX) {
2          super.reset(newX);
3          // Obtenim un número aleatori entre MIN i MAX
4          float newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
               MAX_ASTEROID);
5          // Modificarem l'alçada i l'amplada segons l'aleatori anterior
6          width = height = 34 * newSize;
7          // La posició serà un valor aleatori entre 0 i l'alçada de l'aplicació
               menys l'alçada de l'asteroide
8          position.y =  new Random().nextInt(Settings.GAME_HEIGHT - (int) height)
               ;
9      }
```

Si tornem a executar el joc observarem com ara sí que es van canviant les dimensions i posicions dels asteroides de manera aleatòria.

Treball amb textures

Per fer més atractiva la nostra aplicació assignarem imatges als objectes. Per fer-ho partirem d'un objecte `texture`, que és una imatge carregada a la memòria de la GPU. Els jocs acostumen a fer servir diversos gràfics i es recomana tenir-los tots en un mateix fitxer (figura 2.12) i dividir-los en `TextureRegions`. Així, carregarem una sola imatge a la memòria i la dividirem en parts més petites que són les que representarem per pantalla.

FIGURA 2.12. 'Sprite sheet' del joc



Per fer-ho, cal obrir l'`AssetManager` i afegir els següents atributs de classe:

```
// Sprite Sheet
    public static Texture sheet;

    // Nau i fons
    public static TextureRegion spacecraft, spacecraftDown, spacecraftUp,
        background;

    // Asteroide
    public static TextureRegion[] asteroid;
    public static Animation asteroidAnim;

    // Explosió
    public static TextureRegion[] explosion;
    public static Animation explosionAnim;
```

Podem fer servir l'aplicació *TexturePacker* per tal d'obtenir tots els *sprites* a LibGDX fent servir un objecte `TextureAtlas`, facilitant-nos la generació i lectura d'*sprites*.

Haurem de definir l'*sprite sheet* i crearem `TextureRegion` per a les tres posicions de la nau, per al fons de pantalla i dos *arrays* per a l'asteroide i per a l'explosió, que a més, necessitaran un objecte del tipus `animation`.

El primer que farem és definir l'*sprite sheet*. Cal copiar el fitxer **sheet.png** al directori *assets* del subprojecte android i afegir les següents línies al mètode `load()`:

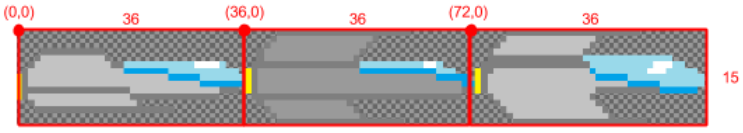
```
// Carreguem les textures i li apliquem el mètode d'escalat 'nearest'
        sheet = new Texture(Gdx.files.internal("sheet.png"));
        sheet.setFilter(Texture.TextureFilter.Nearest, Texture.TextureFilter.
            Nearest);
```

La primera línia definirà el `Texture` i la segona definirà els filtres d'escalat que s'aplicaran en cas de ser necessari: el primer paràmetre fa referència al filtre que s'aplicarà en cas que la imatge s'hagi de disminuir i el segon en cas de necessitar augmentar-se. A la figura 2.13 podeu veure una comparació dels `TextureFilter` *nearest* i *linear*.

Figura 2.13. Diferència entre filtres

Començarem generant els `TextureRegion` de les naus. Hem de calcular la coordenada inicial del rectangle (sempre a la cantonada superior esquerra), l'amplada i l'alçada de l'element, el codi de les naus. Si ens fixem en la figura figura 2.14, quedaria de la següent manera:

Figura 2.14. Càlcul dels TextureRegion




```
// Sprites de la nau
        spacecraft = new TextureRegion(sheet, 0, 0, 36, 15);
        spacecraft.flip(false, true);

        spacecraftUp = new TextureRegion(sheet, 36, 0, 36, 15);
        spacecraftUp.flip(false, true);

        spacecraftDown = new TextureRegion(sheet, 72, 0, 36, 15);
        spacecraftDown.flip(false, true);
```

Al mètode `flip(boolean x, boolean y)` indiquem mitjançant un `true` si volem voltejar la imatge en l'eix horitzontal (X) i en el vertical (Y). En el nostre cas, com que LibGDX fa servir un sistema de coordenades *Y-Up* i nosaltres estem fent servir *Y-Down*, hem de voltejar la imatge en l'eix de les Y fent la crida al mètode `flip(false, true)`.

En el cas de l'asteroide haurem de crear un *array* amb tots els *sprites* de l'animació. A l'*sprite sheet* podem veure com tenim tots els *sprites* en una sola filera i les dimensions de cada element són de 34x34 píxels. Així, haurem de fer 16 iteracions guardant cada *sprite* de l'asteroide. El bucle quedaria així:

```
// Carreguem els 16 estats de l'asteroide
        asteroid = new TextureRegion[16];
        for (int i = 0; i < asteroid.length; i++) {

            asteroid[i] = new TextureRegion(sheet, i * 34, 15, 34, 34);
            asteroid[i].flip(false, true);
```

```
7
8         }
```

Primerament inicialitzem l'*array* amb 16 posicions (de la 0 a la 15), generem el `for` perquè faci iteracions de 0 a 15 i en cadascun calculem la posició de l'*sprite*. Multipliquem el valor d'`i` per 34 per obtenir els valors d'X que seran: 0, 34, 68, 102... Mentre que la Y serà sempre 15 (on acabava la nau espacial). L'amplada i alçada sempre seran 34x34, les dimensions dels asteroides. Per cada iteració, a més, farem el `flip` corresponent per adaptar-la al nostre sistema de coordenades.

Una vegada tenim carregats els *sprites* a l'*array*, haurem de crear l'animació, el constructor ens demana un `float` amb els segons que durarà cada *frame* i l'*array* de *TextureRegion*. Així, amb les línies:

```
1 // Creem l'animació de l'asteroide i fem que s'executi contínuament en sentit
      antihorari
2         asteroidAnim = new Animation(0.05f, asteroid);
3         asteroidAnim.setPlayMode(Animation.PlayMode.LOOP_REVERSED);
```

Crearem i configurarem l'animació perquè es repeteixi de manera invertida (des del final cap al principi: `LOOP_REVERSED`) i així fer que l'asteroide giri en sentit antihorari.

Entre els `PlayMode` tenim:

- `Animation.PlayMode.NORMAL`: l'animació no es repetirà, reproduirà els *frames* del 0 fins el final de l'*array*.
- `Animation.PlayMode.REVERSED`: sense repetir-se, mostrarà els *frames* en ordre invers, des del final fins al *frame* 0.
- `Animation.PlayMode.LOOP`: quan acabi l'animació la tornarà a començar des de l'inici.
- `Animation.PlayMode.LOOP_REVERSED`: farà l'animació en ordre descendent i tornarà a repetir-la en acabar.
- `Animation.PlayMode.LOOP_PINGPONG`: comença l'animació en sentit ascendent i quan arriba al final la fa en sentit descendent i així successivament. Penseu en el joc del ping-pong, en com la pilota va del costat esquerre al dret, del costat dret torna al costat esquerre i tornem a començar de nou.
- `Animation.PlayMode.LOOP_RANDOM`: agafa imatges a l'atzar de l'*array* per crear l'animació.

Seguint els mateixos passos que l'animació dels asteroides, crearem l'animació de l'explosió. Hi ha, però, dues diferències: els *sprites* necessaris es troben en dues files de l'*sprite sheet* i l'animació de l'explosió sols s'executarà una vegada. El primer que hem de fer és pensar de quina manera podem obtenir tots els *sprites* en un *array*: necessitarem fer dos bucles. En el primer controlarem quina fila estem llegint, que serà o la 0 o la 1, i per cadascuna de les files haurem de llegir 8 *sprites*

(segon bucle). Abans de començar els bucles crearem un índex que s'incrementarà de 0 a 15 per guardar cadascun dels *sprites* en l'*array*.

El bucle per guardar els *sprites* de l'explosió seria:

```
// Creem els 16 estats de l'explosió
        explosion = new TextureRegion[16];

        // Carreguem els 16 estats de l'explosió
        int index = 0;
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 8; j++) {
                explosion[index++] = new TextureRegion(sheet, j * 64,  i * 64 +
                     49, 64, 64);
                explosion[index-1].flip(false, true);
            }
        }
```

Les files es calcularan al tercer paràmetre del `TextureRegion`, `i * 64 + 49`: a la primera iteració la Y serà 49 (que era on acabaven els *sprites* dels asteroides) i a la segona iteració serà 113, és a dir, 49 dels asteroides més els 64 de la primera fila de l'explosió. En cadascuna de les files, la X anirà augmentant de 64 en 64, prenent els valors 0, 64, 128,... fins a 448.

Per guardar-les a l'*array* fem servir l'expressió `explosion[index++]`, que seria l'equivalent a realitzar dues accions: `explosion[index]` i després `index += 1`. El `flip` del `TextureRegion` l'haurem de fer al de la posició `index-1` ja que en actualitzar l'índex en la instrucció anterior aquest estarà apuntant al següent valor del bucle.

Finalment creem l'animació amb la línia `explosionAnim = new Animation(0.04f, explosion);` i en aquest cas la imatge no es repetirà així que no serà necessari configurar el `setPlayMode` (el mode per defecte és `PlayMode.NORMAL`).

Finalment, ja sols ens queda afegir el `TextureRegion` del fons de pantalla (480x135 píxels) que quedaria de la següent manera:

```
// Fons de pantalla
        background = new TextureRegion(sheet, 0, 177, 480, 135);
        background.flip(false, true);
```

Al mètode `dispose()` haurem de cridar al `dispose()` de l'element `texture`:

```
public static void dispose() {

        // Descartem els recursos
        sheet.dispose();

    }
```

I amb això podem deixar de banda la classe `AssetManager` per centrar-nos en els objectes `actor`. Començarem per dibuixar el fons de pantalla i la nau espacial.

L'objecte encarregat de dibuixar els *sprites* serà un `batch`. Tots els objectes descendents d'`actor` disposen del mètode `draw(Batch batch, float parentAlpha)` que rep com a argument el `batch` necessari. El mètode `draw()`

s'executarà sempre que s'executi el mètode `draw` de l'`Stage` que els conté així que serà el millor lloc per dibuixar els objectes. Si traiem un `actor` de l'`stage` automàticament deixarà de dibuixar-se de la pantalla i si el tornem a afegir ho tornarà a fer. Això ens estalviarà moltes comprovacions i booleans innecessaris.

Per dibuixar el fons de pantalla haurem d'anar a la classe `Background` i fer l'`Override` del mètode `draw` (*Code/Override Methods...*). Des d'aquí podrem cridar al mètode `batch.draw(Texture texture, float x, float y, float width, float height)` per dibuixar qualsevol textura. Com que volem dibuixar el fons de pantalla, haurem de seleccionar la textura `AssetManager.background` i passem les propietats de l'element:

```
1  public void draw(Batch batch, float parentAlpha) {
2          super.draw(batch, parentAlpha);
3          batch.disableBlending();
4          batch.draw(AssetManager.background, position.x, position.y, width,
               height);
5          batch.enableBlending();
6      }
```

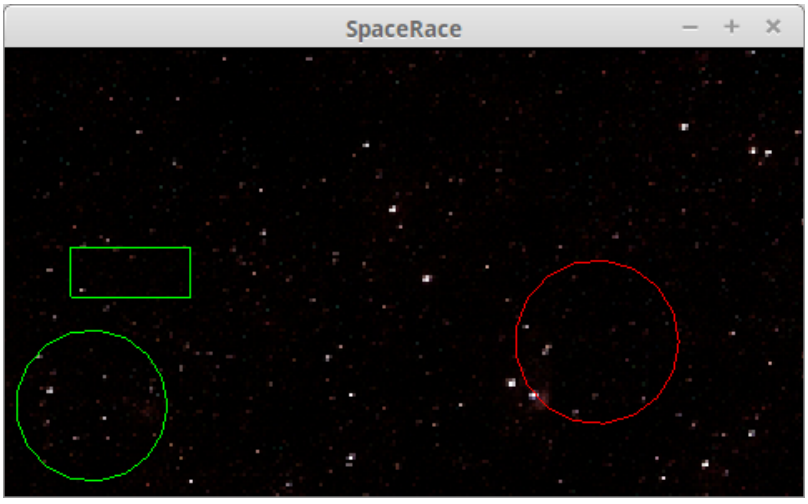
Abans de cridar al mètode `batch.draw()`, executarem `batch.disableBlending()` que ens dibuixarà imatges sense transparència d'una manera més eficient (el `blending` està activat per defecte). Una vegada dibuixat el fons el tornarem a habilitar per no alterar la resta d'objectes `actor` que potser sí que necessiten transparència.

Si comentem les línies:

```
1  //Gdx.gl20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
2          //Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

Del mètode `drawElements()` de la classe `GameScreen` (les encarregades de pintar el fons negre), enexecutar el joc tindrem com a resultat el que podem veure a la figura 2.15.

Figura 2.15. Aplicació amb el fons pintat



Per dibuixar la nau, haurem de fer el mateix. Cal anar a la classe `Spacecraft` i fem l'`Override` del mètode `draw`:

```
@Override
    public void draw(Batch batch, float parentAlpha) {
        super.draw(batch, parentAlpha);
        batch.draw(AssetManager.spacecraft, position.x, position.y, width,
            height);
    }
```

En el cas de la nau espacial no deshabilitem el `blending` ja que necessitem transparència.

Sols ens falta representar els asteroides, però abans haurem de fer unes modificacions en la classe `asteroid` per tal de calcular el *frame* que s'ha de mostrar en cada crida del mètode `draw`. Per fer això necessitem una variable que es vagi incrementant, que anomenarem `runTime`. Cada `asteroid` tindrà un `runTime` que s'actualizarà al seu mètode `act()`.

Obriu la classe `asteroid` i afegiu l'atribut `private float runTime` (amb el corresponent *getter*) que inicialitzarem al constructor amb un valor aleatori entre 0 i 1. Feu un `Override` dels mètodes `draw` i `act`, en `act` actualitzarem el valor del `runTime` incrementant-lo en `delta`. Això ens deixarà una classe com la següent:

```
package cat.xtec.ioc.objects;

import com.badlogic.gdx.graphics.g2d.Batch;

import java.util.Random;

import cat.xtec.ioc.utils.Methods;
import cat.xtec.ioc.utils.Settings;

public class Asteroid extends Scrollable {

    private float runTime;

    public Asteroid(float x, float y, float width, float height, float velocity
        ) {
        super(x, y, width, height, velocity);
        runTime = Methods.randomFloat(0,1);
    }

    @Override
    public void act(float delta) {
        super.act(delta);
        runTime += delta;
    }

    @Override
    public void reset(float newX) {
        super.reset(newX);
        // Obtenim un número al·leatori entre MIN i MAX
        float newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
            MAX_ASTEROID);
        // Modificarem l'alçada i l'amplada segons l'al·leatori anterior
        width = height = 34 * newSize;
        // La posició serà un valor aleatòri entre 0 i l'alçada de l'aplicació
            menys l'alçada
        position.y =  new Random().nextInt(Settings.GAME_HEIGHT - (int) height)
            ;

    }

    // Getter pel runTime
    public float getRunTime() {

```

```
        return runTime;
    }

    @Override
    public void draw(Batch batch, float parentAlpha) {
        super.draw(batch, parentAlpha);
    }
}
```

Per obtenir l'*sprite* de l'animació de cada asteroide, haurem de cridar a `AssetManager.asteroidAnim.getKeyFrame(runTime)`, és a dir, canviarem el *frame* segons la variable `runTime` que s'anirà incrementant. Si afegim el codi al mètode `draw`:

```
@Override
    public void draw(Batch batch, float parentAlpha) {
        super.draw(batch, parentAlpha);
        batch.draw(AssetManager.asteroidAnim.getKeyFrame(runTime), position.x,
            position.y, width, height);
    }
```

Veiem com dibuixarà els asteroides que aniran animant-se a la velocitat definida per l'objecte `asteroidAnim`.

Comenteu la crida a `drawElements()` del mètode `render` de la classe `GameScreen`, executeu el joc a l'ordinador i proveu-lo també a un dispositiu físic si podeu. Ja tenim els gràfics llestos i només ens quedarà provar l'animació de l'explosió que afegirem en l'apartat de col·lisions.

Sons

Treballar amb sons i música en LibGDX és molt senzill. Veurem, ara, un exemple de cada tipus i modificarem el projecte per posar música de fons. El so de l'explosió l'implementarem més endavant en l'apartat de col·lisions.

Per tal d'afegir un so, ho farem mitjançant un objecte `sound`, i cridant el mètode `Gdx.audio.newSound`, per exemple, per afegir el so de l'explosió, que es troba a *android/assets/sounds/explosion.wav* ho farem mitjançant la següent instrucció:

```
explosionSound = Gdx.audio.newSound(Gdx.files.internal("sounds/explosion.wav"))
    ;
```

Cada cop que fem `play()` del so, ens retornarà un identificador (`long`) que necessitarem per realitzar tasques com ara canviar el volum, posar-lo en pausa, reprendre'l, etc. Alguns dels principals mètodes per treballar amb sons són:

- `dispose()`: allibera tots els recursos.
- `loop()`: comença la reproducció del so amb la repetició activada.
- `play()`: comença la reproducció del so, retorna un identificador que ens servirà per controlar la instància del so.
- `pause()`: aturem la reproducció per continuar-la des del mateix punt on l'hem aturat de totes les instàncies del so.

- `pause(long soundId)`: aturem la reproducció per continuar-la des del mateix punt on l'hem aturat d'una instància concreta del so.
- `resume()`: reprèn la reproducció de totes les instàncies del so.
- `resume(long soundId)`: reprèn la reproducció d'una instància del so.
- `stop()`: atura la reproducció de totes les instàncies del so.
- `stop(long soundId)`: atura la reproducció d'una instància del so.
- `setLooping(long soundId, boolean isLooping)`: reprodueix de manera infinita amb `true` i amb una sola reproducció amb `false` d'una instància del so.
- `setVolume(long soundId, float volume)`: ens permet modificar el volum d'una instància concreta del so (valor entre 0.0f i 1.0f).

L'objecte `sound` ens servirà per a sons curts que no necessiten *streaming* i que, a més, puguin tenir diverses instàncies. Si volem afegir un fitxer de més durada i que ocupa més espai al disc, hem de fer servir un objecte `Music`. Per exemple, si volem carregar el fitxer *android/assets/sounds/Afterburner.ogg* hauríem de fer-ho amb:

```
1 music = Gdx.audio.newMusic(Gdx.files.internal("sounds/Afterburner.ogg"));
```

Els objectes `music` només tenen una instància, al contrari que els objectes `sound`. Alguns dels mètodes que podrem fer servir als objectes `music` són:

- `dispose()`: quan l'objecte no es necessita.
- `play()`: comença la reproducció. Si estava pausada la reprèn.
- `pause()`: atura la reproducció per continuar-la des del mateix punt on l'hem aturat.
- `stop()`: atura la reproducció de la música.
- `setLooping(boolean isLooping)`: reprodueix de manera infinita amb `true` i amb una sola reproducció amb `false`.
- `isLooping()`: retorna un booleà indicant si el fitxer de música està o no configurat amb la reproducció infinita.
- `isPlaying()`: retorna un booleà per saber si s'està reproduint o no un fitxer de música.
- `getPosition()`: retorna la posició en segons de la reproducció.
- `setVolume(float volume)`: ens permet modificar el volum de la música (valor entre 0.0f i 1.0f).

Afegirem, ara, música al nostre joc i prepararem el so de l'explosió per més endavant. Per fer-ho, obriu el fitxer `AssetManager` i afegiu les variables de classe `explosionSound` i `music`:

```
// Sons
    public static Sound explosionSound;
    public static Music music;
```

Aquestes variables les inicialitzaren al final del mètode `load()` amb les següents línies:

```
/****************************** Sounds ***********************************/
        // Explosió
        explosionSound = Gdx.audio.newSound(Gdx.files.internal("sounds/
            explosion.wav"));

        // Música del joc
        music = Gdx.audio.newMusic(Gdx.files.internal("sounds/Afterburner.ogg")
            );
        music.setVolume(0.2f);
        music.setLooping(true);
```

Carreguem els dos fitxers del directori *assets/sounds* i configurem la música baixant-li el volum i fent que es repeteixi en acabar. Haurem d'afegir en el mètode `dispose()` les crides a `dispose` dels dos recursos sonors:

```
public static void dispose() {

        // Descartem els recursos
        sheet.dispose();
        explosionSound.dispose();
        music.dispose();
}
```

Per reproduir la música anem a la classe `GameScreen` i afegim al constructor la següent línia:

```
// Iniciem la música
        AssetManager.music.play();
```

2.3.5 Control de la nau

El control de la nau el farem mitjançant la pantalla tàctil: un desplaçament cap a baix farà que la nau incrementi el valor de la Y i farà el contrari en cas d'un desplaçament cap a dalt.

Per gestionar l'entrada de l'usuari anem a *Crea una classe* al paquet `cat.xtec.ioc.helpers`. La classe ha de ser `InputHandler` que implementarà la interfície `InputProcessor`. Per fer-ho, haurem d'implementar els mètodes:

- `keyDown`: s'executa quan es prem una tecla.
- `keyUp`: s'executa quan deixem anar una tecla després d'haver-la premut.
- `keyTyped`: s'executa quan s'escriu un caràcter a un element que té el focus del teclat.

- `touchDown`: s'executa quan l'usuari pressiona en algun lloc de la pantalla.
- `touchUp`: codi que s'executarà quan l'usuari deixi anar el dit de la pantalla.
- `touchDragged`: s'executa quan es produeix un arrossegament del dit per la pantalla.
- `mouseMoved`: exclusiva de les aplicacions d'escriptori, s'executa quan es mou el ratolí sobre la superfície de l'aplicació.
- `scrolled`: exclusiva de les aplicacions d'escriptori, respon a un desplaçament de la roda del ratolí.

Una vegada enllestida la classe tindrà el següent codi:

```
1  package cat.xtec.ioc.helpers;
2
3  import com.badlogic.gdx.InputProcessor;
4
5  public class InputHandler implements InputProcessor {
6
7      @Override
8      public boolean keyDown(int keycode) {
9          return false;
10     }
11
12     @Override
13     public boolean keyUp(int keycode) {
14         return false;
15     }
16
17     @Override
18     public boolean keyTyped(char character) {
19         return false;
20     }
21
22     @Override
23     public boolean touchDown(int screenX, int screenY, int pointer, int button)
           {
24         return false;
25     }
26
27     @Override
28     public boolean touchUp(int screenX, int screenY, int pointer, int button) {
29         return false;
30     }
31
32     @Override
33     public boolean touchDragged(int screenX, int screenY, int pointer) {
34         return false;
35     }
36
37     @Override
38     public boolean mouseMoved(int screenX, int screenY) {
39         return false;
40     }
41
42     @Override
43     public boolean scrolled(int amount) {
44         return false;
45     }
46 }
```

Els tres *events* que ens interessen en aquest moment són: `touchDown()`, `touchUp()` i `touchDragged()`. Quan l'usuari arrossegui el dit canviarem la

direcció de la nau i quan l'aixequi la nau deixarà de moure's. Necessitarem l'objecte `spacecraft` i l'objecte `GameScreen`, que el rebrem com a paràmetre del constructor, des del qual podrem accedir a la nau.

Afegirem a la classe les variables d'instància i crearem el constructor. També caldrà afegir un enter que ens permetrà saber en quina direcció s'ha fet un moviment de *drag*:

```
// Objectes necessaris
    private Spacecraft spacecraft;
    private GameScreen screen;

    // Enter per a la gestió del moviment d'arrossegament
    int previousY = 0;

    public InputHandler(GameScreen screen) {

        // Obtenim tots els elements necessaris
        this.screen = screen;
        spacecraft = screen.getSpacecraft();

    }
```

Ja ho tenim tot preparat per controlar quan l'usuari arrossega un dit. El mètode `touchDragged` rep tres paràmetres:

- `int screenX`: coordenada horitzontal (X) del dit.
- `int screenY`: coordenada vertical (Y) del dit.
- `int pointer`: identificador del punter actiu.

Com que volem desplaçar la nau en l'eix d'Y, necessitarem utilitzar únicament el paràmetre `screenY`: si `screenY` és major que l'anterior Y vol dir que s'està arrossegant el dit cap avall (sistema *Y-Down*) i per tant la nau haurà de baixar. En cas contrari haurà de pujar. Guardarem la posició quan l'usuari faci clic a la pantalla i caldrà definir un llindar per acceptar un moviment i evitar que tenint el dit quiet la nau es mogui cap a un o altre costat. Posarem un llindar de 2 píxels.

El codi corresponent als mètodes `touchDragged` i `touchDown` serà el següent:

```
@Override
    public boolean touchDown(int screenX, int screenY, int pointer, int button)
         {
        previousY = screenY;
        return true;
    }

    @Override
    public boolean touchDragged(int screenX, int screenY, int pointer) {
        // Posem un llindar per evitar gestionar events quan el dit està quiet
        if (Math.abs(previousY - screenY) > 2)

            // Si la Y és major que la que tenim
            // guardada és que va cap avall
            if (previousY < screenY) {
                spacecraft.goDown();
            } else {
                // En cas contrari cap amunt
                spacecraft.goUp();
```

```
            }
        // Guardem la posició de la Y
        previousY = screenY;
        return true;
    }
```

Hem d'assignar la classe `InputHandler` com a `InputProcessor` de la classe `GameScreen`. Cal anar al constructor de `GameScreen` i afegir al final les línies:

```
// Assignem com a gestor d'entrada la classe InputHandler
        Gdx.input.setInputProcessor(new InputHandler(this));
```

Si executem l'aplicació podrem desplaçar la nau en vertical però no la podrem aturar. Per fer-ho hauríem d'implementar el mètode `touchUp` i fer que la nau torni al seu estat natural quan s'aixequi el dit:

```
@Override
    public boolean touchUp(int screenX, int screenY, int pointer, int button) {

        // Quan deixem anar el dit acabem un moviment
        // i posem la nau a l'estat normal
        spacecraft.goStraight();
        return true;
    }
```

Amb aquests mètodes ja tenim la funcionalitat implementada, però la representació de la nau no és la correcta, hem de fer que mostri l'*sprite* corresponent quan la nau està movent-se. Anem a la classe `spacecraft` i afegim un mètode que retornarà el `TextureRegion` corresponent segons la direcció de la nau, i farem la crida a aquest mètode quan anem a dibuixar-la.

El mètode `getSpacecraftTexture()` serà el següent:

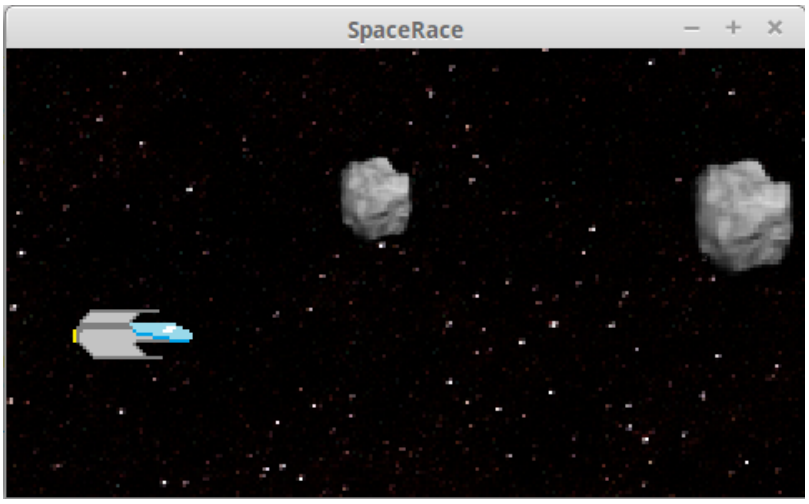
```
// Obtenim el TextureRegion depenent de la posició de l'spacecraft
    public TextureRegion getSpacecraftTexture() {

        switch (direction) {

            case SPACECRAFT_STRAIGHT:
                return AssetManager.spacecraft;
            case SPACECRAFT_UP:
                return AssetManager.spacecraftUp;
            case SPACECRAFT_DOWN:
                return AssetManager.spacecraftDown;
            default:
                return AssetManager.spacecraft;
        }
    }
```

I el mètode `draw` el modificarem perquè quedi de la següent manera:

```
@Override
    public void draw(Batch batch, float parentAlpha) {
        super.draw(batch, parentAlpha);
        batch.draw(getSpacecraftTexture(), position.x, position.y, width,
            height);
    }
```

Si executem l'aplicació ja tindrem el moviment de la nau i la representació correctament implementats, tal com podem veure a la figura 2.16.

FIGURA 2.16. Nau desplaçant-se



2.3.6 Col·lisions

Hi ha diverses maneres de gestionar les col·lisions. Existeixen llibreries externes com per exemple `Box2D` que disposa d'un gestor de col·lisions. També és possible fer-ho comprovant la intersecció entre figures geomètriques. Pel nostre joc, podrem comprovar les col·lisions entre la nau (rectangle) i els asteroides (cercles) amb el mètode `Intersector.overlaps` que accepta diverses figures geomètriques, com per exemple:

- `Intersector.overlaps(Circle c, Recangle r)`: retorna `true` si es sobreposen un cercle i un rectangle.
- `Intersector.overlaps(Circle c1, Circle c2)`: retorna `true` si es sobreposen dos cercles.
- `Intersector.overlaps(Rectangle r1, Recangle r2)`: retorna `true` si es sobreposen dos rectangles.
- `Intersector.overlapConvexPolygons(Polygon p1, Polygon p2)`: retorna `true` si es sobreposen dos polígons.

Necessitem llavors un rectangle per a la nau i cercles per a cadascun dels asteroides.

Anem a la classe `Spacecraft` i creem una variable d'instància `private Rectangle collisionRect;` que serà un `Rectangle` de `com.badlogic.gdx.math.Rectangle`. Aquest `Rectangle` l'haurem d'anar actualitzant segons la posició de la nau. Creem el rectangle i actualitzem la seva posició en el mètode `act` obtenint com a resultat el codi del constructor i del mètode `act` de la següent manera:

```
1 public Spacecraft(float x, float y, int width, int height) {
2
3         // Inicialitzem els arguments segons la crida del constructor
```

```
        this.width = width;
        this.height = height;
        position = new Vector2(x, y);

        // Inicialitzem l'spacecraft a l'estat normal
        direction = SPACECRAFT_STRAIGHT;

        // Creem el rectangle de col·lisions
        collisionRect = new Rectangle();

    }

    public void act(float delta) {

        // Movem l'spacecraft depenent de la direcció controlant que no surti
            de la pantalla
        switch (direction) {
            case SPACECRAFT_UP:
                if (this.position.y - Settings.SPACECRAFT_VELOCITY * delta >=
                    0) {
                    this.position.y -= Settings.SPACECRAFT_VELOCITY * delta;
                }
                break;
            case SPACECRAFT_DOWN:
                if (this.position.y + height + Settings.SPACECRAFT_VELOCITY *
                    delta <= Settings.GAME_HEIGHT) {
                    this.position.y += Settings.SPACECRAFT_VELOCITY * delta;
                }
                break;
            case SPACECRAFT_STRAIGHT:
                break;
        }

        collisionRect.set(position.x, position.y + 3, width, 10);

    }
```

En el codi anterior s'ha ajustat el rectangle 3 píxels en vertical i hem modificat l'alçada per evitar els espais transparents del `TextureRegion`.

Afegim també el corresponent *getter*.

```
public Rectangle getCollisionRect() {
        return collisionRect;
    }
```

Pel que fa als `asteroid`, haurem de crear un cercle i actualitzar-lo cada vegada que es mou l'objecte. Per fer-ho cal anar a la classe`asteroid` i crear la variable d'instància `private Circle collisionCircle;` que serà un `com.badlogic.gdx.math.Circle`. L'iniciem i actualitzem deixant el constructor i el mètode `act` de la següent manera:

```
public Asteroid(float x, float y, float width, float height, float velocity) {
        super(x, y, width, height, velocity);
        runTime = Methods.randomFloat(0,1);

        // Creem el cercle
        collisionCircle = new Circle();
    }

    @Override
    public void act(float delta) {
        super.act(delta);
        runTime += delta;

```

```
        // Actualitzem el cercle de col·lisions (punt central de l'asteroide i
            del radi).
        collisionCircle.set(position.x + width / 2.0f, position.y + width / 2.0
            f, width / 2.0f);

    }
```

Des del mètode `render` de la classe `GameScreen`, anirem comprovant si els objectes `Scrollable` col·lideixen amb la nau. Per fer-ho hem de cridar al mètode `scrollhandler.collides(spacecraft)`: si el mètode citat torna `true`, llavors farem explotar i desaparèixer la nau:

```
@Override
    public void render(float delta) {

        // Dibuixem i actualitzem tots els actors de l'stage
        stage.draw();
        stage.act(delta);

        if (scrollHandler.collides(spacecraft)) {

            // La nau explota i desapareix
            Gdx.app.log("App", "Explosió");

        }

        //drawElements();

    }
```

Haurem de crear el mètode `collides` a la classe `ScrollHandler`. Aquest mètode comprovarà si algun dels asteroides col·lideixen amb la nau i en cas de que algun ho faci retornarem `true`, afegim el mètodes `collides`:

```
public boolean collides(Spacecraft nau) {

        // Comprovem les col·lisions entre cada asteroide i la nau
        for (Asteroid asteroid : asteroids) {
            if (asteroid.collides(nau)) {
                return true;
            }
        }
        return false;
    }
```

I sols ens faltarà el mètode `collides` de la classe `asteroid` on comprovarem si el cercle de l'asteroide se sobreposa al rectangle de la nau sempre que l'asteroide es troba a l'alçada de la nau:

```
// Retorna true si hi ha col·lisió
    public boolean collides(Spacecraft nau) {

        if (position.x <= nau.getX() + nau.getWidth()) {
            // Comprovem si han col·lisionat sempre que l'asteroide es trobi a
                la mateixa alçada que l'spacecraft
            return (Intersector.overlaps(collisionCircle, nau.getCollisionRect
                ()));
        }
        return false;
    }
```

Executem l'aplicació i veurem com cada vegada que es produeix una col·lisió apareixen missatges en el *log* de l'Android Studio.

Si volem controlar quan s'ha de deixar de comprovar les col·lisions, necessitarem un booleà `gameOver` inicialment a `false`. Si `gameOver` és `false` i s'ha produït una col·lisió:

- Reproduïm el so de l'explosió: `AssetManager.explosionSound.play();`
- Eliminem l'`actor` de l'`stage` perquè no continuï actualitzant-se ni dibuixant-se per pantalla. Per fer-ho, necessitarem fer dos passos:
  1. Posar-li nom a l'`actor` després d'afegir-lo a l'`stage` amb la instrucció `spacecraft.setName("spacecraft")`.
  2. Cercar-lo entre els actors de l'`stage` amb `stage.getRoot().findActor("spacecraft")` i fent la crida al mètode `remove()`.
- Posem la variable `gameOver` a `false`.

Si `gameOver` és `true`:

- Dibuixem l'explosió.
- Incrementem la variable `explosionTime` (prèviament declarada i inicialitzada a 0) sumant-li el valor `delta`.

El mètode `render` quedarà de la següent manera:

```
@Override
    public void render(float delta) {

        // Dibuixem i actualitzem tots els actors de l'stage
        stage.draw();
        stage.act(delta);

        if (!gameOver) {
            if (scrollHandler.collides(spacecraft)) {
                // Si hi ha hagut col·lisió: Reproduïm l'explosió
                AssetManager.explosionSound.play();
                stage.getRoot().findActor("spacecraft").remove();
                gameOver = true;
            }
        } else {
            batch.begin();
            // Si hi ha hagut col·lisió: reproduïm l'explosió
            batch.draw(AssetManager.explosionAnim.getKeyFrame(explosionTime,
                 false), (spacecraft.getX() + spacecraft.getWidth() / 2) - 32,
                 spacecraft.getY() + spacecraft.getHeight() / 2 - 32, 64, 64);
            batch.end();

            explosionTime += delta;
        }

        //drawElements();

    }
```

I la classe `GameScreen`:

```
package cat.xtec.ioc.screens;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.Batch;
import com.badlogic.gdx.graphics.glutils.ShapeRenderer;
import com.badlogic.gdx.scenes.scene2d.Stage;
import com.badlogic.gdx.utils.viewport.StretchViewport;

import java.util.ArrayList;

import cat.xtec.ioc.helpers.AssetManager;
import cat.xtec.ioc.helpers.InputHandler;
import cat.xtec.ioc.objects.Asteroid;
import cat.xtec.ioc.objects.ScrollHandler;
import cat.xtec.ioc.objects.Spacecraft;
import cat.xtec.ioc.utils.Settings;

public class GameScreen implements Screen {

    // Per controlar el gameover
    Boolean gameOver = false;

    // Objectes necessaris
    private Stage stage;
    private Spacecraft spacecraft;
    private ScrollHandler scrollHandler;

    // Encarregats de dibuixar elements per pantalla
    private ShapeRenderer shapeRenderer;
    private Batch batch;

    // Per controlar l'animació de l'explosió
    private float explosionTime = 0;

    public GameScreen() {

        // Iniciem la música
        AssetManager.music.play();

        // Creem el ShapeRenderer
        shapeRenderer = new ShapeRenderer();

        // Creem la càmera de les dimensions del joc
        OrthographicCamera camera = new OrthographicCamera(Settings.GAME_WIDTH,
                Settings.GAME_HEIGHT);
        // Posant el paràmetre a true configurem la càmera perquè
        // faci servir el sistema de coordenades Y-Down
        camera.setToOrtho(true);

        // Creem el viewport amb les mateixes dimensions que la càmera
        StretchViewport viewport = new StretchViewport(Settings.GAME_WIDTH,
            Settings.GAME_HEIGHT, camera);

        // Creem l'stage i assginem el viewport
        stage = new Stage(viewport);

        batch = stage.getBatch();

        // Creem la nau i la resta d'objectes
        spacecraft = new Spacecraft(Settings.SPACECRAFT_STARTX, Settings.
            SPACECRAFT_STARTY, Settings.SPACECRAFT_WIDTH, Settings.
            SPACECRAFT_HEIGHT);
        scrollHandler = new ScrollHandler();

        // Afegim els actors a l'stage
```

```
65          stage.addActor(scrollHandler);
66          stage.addActor(spacecraft);
67          // Donem nom a l'Actor
68          spacecraft.setName("spacecraft");
69
70          // Assignem com a gestor d'entrada la classe InputHandler
71          Gdx.input.setInputProcessor(new InputHandler(this));
72
73      }
74
75      private void drawElements() {
76
77          // Recollim les propietats del batch de l'stage
78          shapeRenderer.setProjectionMatrix(batch.getProjectionMatrix());
79
80          // Pintem el fons de negre per evitar el "flickering"
81          //Gdx.gl20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
82          //Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
83
84          // Inicialitzem el shaperenderer
85          shapeRenderer.begin(ShapeRenderer.ShapeType.Line);
86
87          // Definim el color (verd)
88          shapeRenderer.setColor(new Color(0, 1, 0, 1));
89
90          // Pintem la nau
91          shapeRenderer.rect(spacecraft.getX(), spacecraft.getY(), spacecraft.
                getWidth(), spacecraft.getHeight());
92
93          // Recollim tots els Asteroid
94          ArrayList<Asteroid> asteroids = scrollHandler.getAsteroids();
95          Asteroid asteroid;
96
97          for (int i = 0; i < asteroids.size(); i++) {
98
99              asteroid = asteroids.get(i);
100             switch (i) {
101                 case 0:
102                     shapeRenderer.setColor(1, 0, 0, 1);
103                     break;
104                 case 1:
105                     shapeRenderer.setColor(0, 0, 1, 1);
106                     break;
107                 case 2:
108                     shapeRenderer.setColor(1, 1, 0, 1);
109                     break;
110                 default:
111                     shapeRenderer.setColor(1, 1, 1, 1);
112                     break;
113             }
114             shapeRenderer.circle(asteroid.getX() + asteroid.getWidth() / 2,
                    asteroid.getY() + asteroid.getWidth() / 2, asteroid.getWidth()
                     / 2);
115         }
116         shapeRenderer.end();
117     }
118
119
120     @Override
121     public void show() {
122
123     }
124
125     @Override
126     public void render(float delta) {
127
128         // Dibuixem i actualizem tots els actors de l'stage
129         stage.draw();
130         stage.act(delta);
131
```

```
132         if (!gameOver) {
133             if (scrollHandler.collides(spacecraft)) {
134                 // Si hi ha hagut col·lisió: reproduïm l'explosió
135                 AssetManager.explosionSound.play();
136                 stage.getRoot().findActor("spacecraft").remove();
137                 gameOver = true;
138             }
139         } else {
140             batch.begin();
141             // Si hi ha hagut col·lisió: reproduïm l'explosió
142             batch.draw(AssetManager.explosionAnim.getKeyFrame(explosionTime,
                        false), (spacecraft.getX() + spacecraft.getWidth() / 2) - 32,
                        spacecraft.getY() + spacecraft.getHeight() / 2 - 32, 64, 64);
143             batch.end();
144 
145             explosionTime += delta;
146         }
147 
148         //drawElements();
149 
150     }
151 
152     @Override
153     public void resize(int width, int height) {
154 
155     }
156 
157     @Override
158     public void pause() {
159 
160     }
161 
162     @Override
163     public void resume() {
164 
165     }
166 
167     @Override
168     public void hide() {
169 
170     }
171 
172     @Override
173     public void dispose() {
174 
175     }
176 
177     public Spacecraft getSpacecraft() {
178         return spacecraft;
179     }
180 
181     public Stage getStage() {
182         return stage;
183     }
184 
185     public ScrollHandler getScrollHandler() {
186         return scrollHandler;
187     }
188 }
```

2.3.7 Text

Un dels elements més importants en els jocs és el text, bé sigui per donar instruccions de com es juga, per avisar el jugador de quan ha acabat el joc, de quina puntuació té, per representar diàlegs de personatges, etc.

LibGDX ens permet generar textos amb la font predeterminada o afegir les nostres pròpies fonts d'una manera senzilla. A més, tenim diverses maneres d'introduir el text per la pantalla, a través del mètode `draw` de la font o bé fent ús de les etiquetes (*label*). Anem a veure exemples dels dos mètodes.

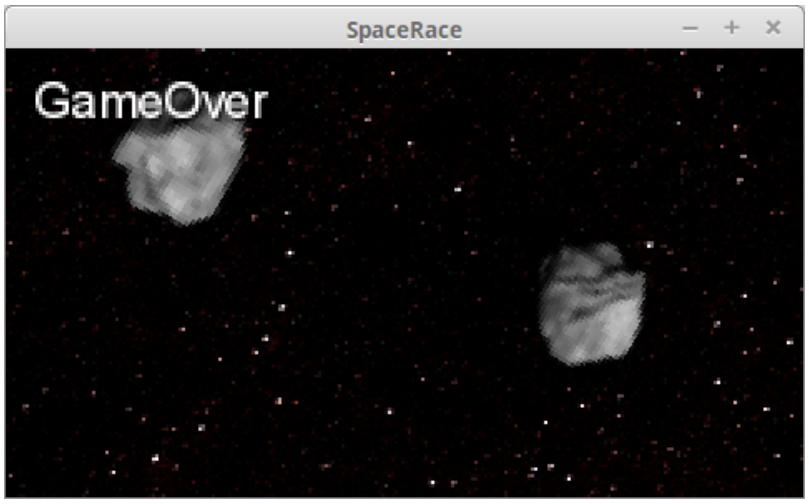
Mètode draw de font

Si volem generar un text molt bàsic, podem crear un nou `BitmapFont(Boolean flip)` i fer la crida al seu mètode `draw(Batch batch, String text, int x, int y)`, i proporcionar-li el `batch`, el text i la coordenada on es mostrarà. Un exemple podria ser:

```
1 BitmapFont font = new BitmapFont(true);
2 font.draw(batch, "GameOver", 10, 10);
```

El paràmetre `true` indica que estem fent servir un sistema *Y-down* i volem girar el text. Si posem aquest text entre `batch.begin()` i `batch.end()` del mètode `render` de la classe `GameScreen` apareixerà el missatge "GameOver" quan col·lideixen contra un asteroide (figura 2.17).

Figura 2.17. Text GameOver en col·lidir amb un asteroide



Podem modificar els paràmetres del text amb mètodes de l'objecte `BitmapFont`. Per exemple, podem canviar el color amb `setColor(Color color)` o `setColor(float r, float g, float b, float a)` i canviar-li la mida al text amb `getData().setScale(float xy)` o `getData().scale(float xy)`. Cal anar amb compte perquè els dos mètodes d'escalat no són iguals, `setScale` defineix la mida, i `scale` aplica una transformació sobre l'escala actual. Per exemple, si posem `setScale(0.25f)` l'objecte serà un 25% de la seva mida original, és a dir, més petit. En canvi, si fem un `scale(0.25f)` estarem fent l'objecte un 25% més gran respecte la mida que tenia.

Per exemple, el codi:

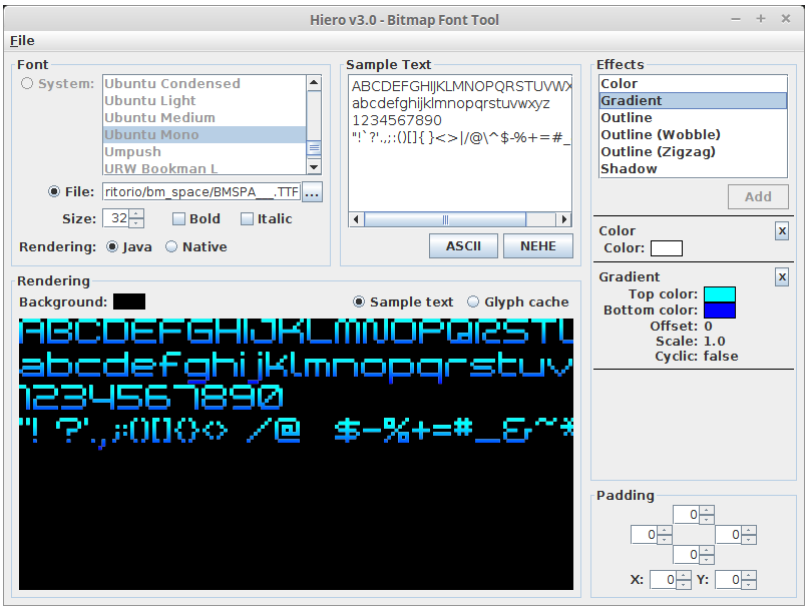
```
1 font.getData().setScale(0.75f);
2 font.setColor(205f/255f, 220f/255f, 57f/255f, 1);
```

Aquest codi fa que el text sigui un 75% de la seva mida original i defineix el color hexadecimal `#CDDC39`, que és: `CD` de vermell (205 en decimal), `DC` de verd (220 en decimal) i `39` de blau (57 en decimal) sense cap tipus de transparència. Dividim entre `FF` (el número màxim amb 2 bits d'hexadecimal que és 255 en decimal) per obtenir el percentatge de cada color.

Si volem personalitzar la font, des de LibGDX ens proporcionen l'eina *Hiero*. La podem instal·lar des de l'assistent de *setup* del projecte (dins del paquet *tools*) o la podem descarregar de: https://github.com/libgdx/libgdx/wiki/Hiero. Anem al directori de descàrrega i executem la següent comanda:

```
ubuntu@ubuntu ~/Descargas$ java –jar hiero.jar
```

Figura 2.18. Eina Hiero configurada



Des de l'aplicació podem seleccionar qualsevol font del sistema o seleccionar un fitxer TTF. Als recursos disposareu de la font *BM_space* per descarregar. Seleccioneu-la des de l'aplicació i proveu-ne diferents configuracions. Per a l'exemple afegirem un gradient fent clic en *Gradient* i a *Add*. Podeu personalitzar els colors del gradient al vostre gust. Podeu veure la captura de l'aplicació en la figura figura 2.18

Una vegada configurada la font, fem clic a *File/Save BMFont Files(text)...* i la guardem en el directori *assets/fonts/* del projecte amb el nom de **space.fnt**.

Després, cal carregar i configurar la font des de la classe `AssetManager`, obrir-la i afegir-hi una variable estàtica per a la font:

```
// Font
    public static BitmapFont font;
```

I al final del mètode load() el següent codi:

```
/****************************** Text ***********************************/
        // Font space
```

```
3         FileHandle fontFile = Gdx.files.internal("fonts/space.fnt");
4         font = new BitmapFont(fontFile, true);
5         font.getData().setScale(0.4f);
```

Si eliminem el text "GameOver" que havíem escrit abans i afegim la següent línia:

```
1 AssetManager.font.draw(batch, "GameOver", 10, 10);
```

S'escriurà el text amb la nova font, però mesurant la meitat que l'original.

El problema el tenim en intentar centrar el text. Podríem pensar que la línia `AssetManager.font.draw(batch, ''GameOver'', Settings.GAME_WIDTH/2, Settings.GAME_HEIGHT/2)` mostrarà el text centrat però no és així, ja que amb les coordenades estem especificant la cantonada superior esquerra d'allà on anirà el text. Per calcular les coordenades necessitem conèixer l'amplada i l'alçada del text. En versions anteriors de LibGDX estava disponible el mètode `BitmapFont.getBounds(String text)` que ens permetia saber les dimensions d'un text però ara no el trobem disponible. Si volem saber les dimensions d'un text haurem de recórrer a un objecte `GlyphLayout`.

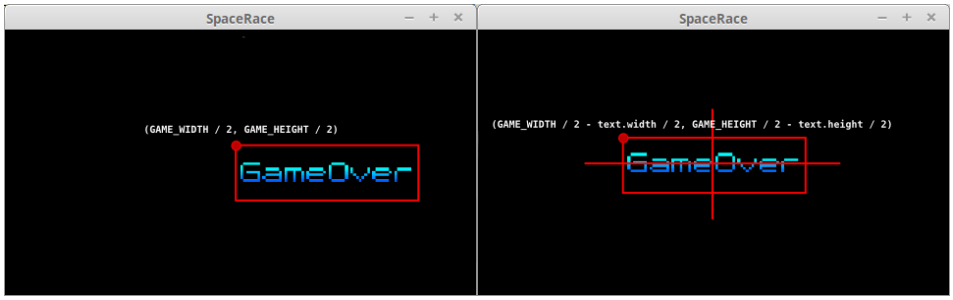
De `GlyphLayout` ens interessa el mètode `setText` que ens demanarà dos paràmetres: la font del text per poder-ne calcular les dimensions i el text. Una vegada crear el *layout* i una vegada li hem assignat el text podrem accedir a les propietats `width` i `height` per saber les dimensions del text.

Després cal centrar el text "GameOver". Per a això hem de crear a `GameScreen` la variable `private GlyphLayout textLayout;` que iniciarem al constructor amb `textLayout = new GlyphLayout();` i assignarem el text "GameOver" amb: `textLayout.setText(AssetManager.font, ''GameOver'');`. Reemplacem la línia `AssetManager.font.draw(batch, ''GameOver'', Settings.GAME_WIDTH/2, Settings.GAME_HEIGHT/2);` de `render` per:

```
1 AssetManager.font.draw(batch, textLayout, Settings.GAME_WIDTH/2 - textLayout.
      width/2, Settings.GAME_HEIGHT/2 - textLayout.height/2);
```

Obtenim el punt central de la pantalla i restem la meitat de l'amplada del text i la meitat de l'alçada per tal que el text quedi centrat.

FIGURA 2.19. Configuracions de posició del text



Sempre que vulguem modificar el contingut del text no hem de fer més que cridar al mètode `setText`.

Objecte label

Els objectes `com.badlogic.gdx.scenes.scene2d.ui.Label` són descendents de `com.badlogic.gdx.scenes.scene2d.ui.Widget` que a la vegada són descendents de `com.badlogic.gdx.scenes.scene2d.Actor`, és a dir, els objectes `label` són a la vegada objectes `actor`.

El fet que siguin `actor` fa que puguem afegir-los a un `stage` i modificar-los de la mateixa manera que fèiem amb la resta d'objectes.

Per a la creació d'un objecte `label` necessitem disposar d'un objecte `LabelStyle` amb la `BitmapFont` i el `color` en què es mostrarà l'etiqueta. Per exemple:

```
1 Label.LabelStyle textStyle = new Label.LabelStyle(AssetManager.font, null);
```

Posem el color a `null` per no modificar-lo. Una vegada tenim l'estil ja podem crear el `label` (text i estil). Per mostrar-lo per pantalla només haurem d'afegir-lo com a `actor` en l'objecte `stage`:

```
1 Label textLbl = new Label("SpaceRace", textStyle);
2 textLbl.setName("lbl");
3 textLbl.setPosition(Settings.GAME_WIDTH/2 - textLbl.getWidth()/2,  Settings.
      GAME_HEIGHT/2 - textLbl.getHeight()/2);
4 stage.addActor(textLbl);
```

Aquest codi crearia el text "SpaceRace" i el posicionaria al centre de la pantalla. Per treure'l hauríem d'eliminar l'actor de l'`stage` amb:

```
1 stage.getRoot().findActor("lbl").remove();
```

2.3.8 Preferències

Tant en les aplicacions d'escriptori com en les de qualsevol plataforma necessitem guardar informació de manera persistent. La manera d'emmagatzemar informació amb LibGDX és molt similar a com es pot fer amb Android.

Per a la gestió de les preferències es necessita un objecte del tipus `com.badlogic.gdx.Preferences`: `Preferences prefs;` per exemple. Aquest objecte l'inicialitzarem indicant un nom de preferències, per exemple: `prefs = Gdx.app.getPreferences(‘‘prefs’’);`. A partir d'aquest moment ja podrem guardar dades fent servir els mètodes `putString`, `pubInteger`, `putBoolean`, `putFloat` i `putLong` o recollir-les amb els corresponents `getString`, `getInteger`, `getBoolean`, `getFloat` o `getLong`.

> Si volem que la informació sigui persistent entre diverses execucions del programa hem de cridar al mètode `flush()` de les preferències.

2.3.9 Actions

Els objectes `actor` són elements molt importants en la programació de jocs amb LibGDX ja que qualsevol objecte pot ser un `actor` de la nostra escena i, per tant, tindrà una sèrie de propietats i mètodes disponibles. A més, LibGDX ens permetrà aplicar `actions` a tots els objectes descendents d'`actor`. Aquestes accions ens proporcionaran una gran versatilitat i ens permetran canviar propietats de l'element d'una forma senzilla.

> Per tal d'aplicar accions a un objecte `label`, tot i ser descendent de la classe `actor`, aquest ha d'estar a dins d'un objecte `table` (si hi ha més elements) o `containner` (si volem aplicar-les solament a l'etiqueta) i les propietats s'han d'aplicar sobre aquest objecte pare. L'estratègia a seguir serà la següent: creem l'etiqueta, creem el contenidor, creem les accions i les apliquem sobre el contenidor. Un `container` hereta de la classe `WidgetGroup`, que a la vegada hereta de `group` i aquesta de la classe `actor`.

El llistat de totes les accions possibles el podeu trobar a la documentació de Package com.badlogic.gdx.scenes.scene2d.actions. Entre les accions més destacades trobem:

- `AlphaAction`: modifica el valor d'*alpha* de l'actor.
- `ColorAction`: canvia el color de l'actor.
- `MoveToAction`: mou l'actor a una determinada posició.
- `MoveByAction`: mou l'actor uns valors determinats respecte la posició actual.
- `RotateToAction`: gira l'actor a un angle concret.
- `RotateByAction`: gira l'actor un angle determinat respecte l'angle actual.
- `ScaleToAction`: escala l'objecte a unes mides concretes.
- `ScaleByAction`: escala l'objecte unes mides determinades respecte la mida actual.
- etc.

I podem utilitzar accions a les pròpies accions, i permetre:

- `ParallelAction`: permet executar accions de manera paral·lela.
- `SequenceAction`: s'executen les accions una rere l'altra.
- `RepeatAction`: per repetir les accions diverses vegades. Serà útil la variable `RepeatAction.FOREVER`.

- `DelayAction`: endarrereix l'execució de l'acció un determinat temps.

> Per poder aplicar una acció a un `actor` que té una classe pròpia és necessari que el mètode `act(float delta)` faci una crida a `super.act(delta)`.

Canviarem l'animació dels asteroides per fer-la mitjançant accions. Així, necessitarem definir una acció per canviar l'angle de l'asteroide i que aquesta acció es faci de manera infinita: `RotateByAction` serà l'encarregada de fer la rotació i `RepeatAction` farà que aquesta es repeteixi infinitament.

Comprovem, ara, que tant les classes `ScrollHandler`, `scrollable` i `asteroid` tenen la sentència `super.act(delta)` en el seu mètode `act()` i en cas de no ser així les afegim. Una vegada fet, en el constructor de l'asteroide crearem les accions i les aplicarem a l'actor.

Afegim l'acció de rotació:

```
1 // Rotació
2         RotateByAction rotateAction = new RotateByAction();
3         rotateAction.setAmount(-90f);
4         rotateAction.setDuration(0.2f);
```

Creem l'acció i diem que giri 90º en sentit antihorari en 0.2 segons. Si volem que es faci de manera infinita, haurem de crear l'acció `RepeatAction`:

```
1 // Accio de repetició
2         RepeatAction repeat = new RepeatAction();
3         repeat.setAction(rotateAction);
4         repeat.setCount(RepeatAction.FOREVER);
```

Creem l'acció, l'assignem a l'acció de rotació i li diem que es repeteixi per sempre. Ja sols ens queda aplicar-la a l'objecte `actor` amb:

```
1 this.addAction(repeat);
```

Li assignem a l'objecte actual (qualsevol `asteroid` que creem) l'acció de repetició. Per dibuixar ara l'element haurem de canviar la línia del mètode `draw()` perquè dibuixi un sol `TextureRegion` amb les seves propietats:

```
1 batch.draw(AssetManager.asteroid[0], position.x, position.y, this.getOriginX(),
      this.getOriginY(), width, height, this.getScaleX(), this.getScaleY(),
     this.getRotation());
```

Si executem l'aplicació veurem que passa una cosa estranya, els asteroides no giren com esperàvem. Això és perquè el seu punt d'origen està definit en la cantonada superior-esquerra. Per solucionar aquest problema haurem de definir l'origen i posar-lo en la part central de la imatge, això ho farem amb `this.setOrigin(width/2, height/2);`. Cada cop que canviem les dimensions dels asteroides cal fer també aquest canvi, per tant crearem un mètode `setOrigin()` que realitzarà l'acció. A més, corregim ara un petit defecte que fa que sembli que la rotació dels *sprites* no és perfecta sumant-li 1 a `width/2` .

Afegim el mètode:

```
public void setOrigin() {

        this.setOrigin(width/2 + 1, height/2);

    }
```

I el cridem des del constructor i des del mètode `reset` ja que s'haurà de recalcular el punt central a causa del canvi de dimensions.

Eliminem tota referència a `runTime` perquè no són necessàries i definim una variable aleatòria (`assetAsteroid`) que agafi els diferents *sprites* de l'*array* d'asteroides. El codi de la classe quedarà així:

```
package cat.xtec.ioc.objects;

import com.badlogic.gdx.graphics.g2d.Batch;
import com.badlogic.gdx.math.Circle;
import com.badlogic.gdx.math.Intersector;
import com.badlogic.gdx.scenes.scene2d.actions.RepeatAction;
import com.badlogic.gdx.scenes.scene2d.actions.RotateByAction;

import java.util.Random;

import cat.xtec.ioc.helpers.AssetManager;
import cat.xtec.ioc.utils.Methods;
import cat.xtec.ioc.utils.Settings;

public class Asteroid extends Scrollable {

    private Circle collisionCircle;

    Random r;

    int assetAsteroid;

    public Asteroid(float x, float y, float width, float height, float velocity
        ) {
        super(x, y, width, height, velocity);

        // Creem el cercle
        collisionCircle = new Circle();

        /* Accions */
        r = new Random();
        assetAsteroid = r.nextInt(15);

        setOrigin();

        // Rotació
        RotateByAction rotateAction = new RotateByAction();
        rotateAction.setAmount(-90f);
        rotateAction.setDuration(0.2f);

        // Acció de repetició
        RepeatAction repeat = new RepeatAction();
        repeat.setAction(rotateAction);
        repeat.setCount(RepeatAction.FOREVER);

        this.addAction(repeat);

    }

    public void setOrigin() {

        this.setOrigin(width/2 + 1, height/2);
```

```
52
53      }
54
55      @Override
56      public void act(float delta) {
57          super.act(delta);
58
59          // Actualitzem el cercle de col·lisions (punt central de l'asteroide i
                el radi.
60          collisionCircle.set(position.x + width / 2.0f, position.y + width / 2.0
                f, width / 2.0f);
61
62
63      }
64
65      @Override
66      public void reset(float newX) {
67          super.reset(newX);
68          // Obtenim un número aleatori entre MIN i MAX
69          float newSize = Methods.randomFloat(Settings.MIN_ASTEROID, Settings.
                MAX_ASTEROID);
70          // Modificarem l'alçada i l'amplada segons l'aleatori anterior
71          width = height = 34 * newSize;
72          // La posició serà un valor aleatori entre 0 i l'alçada de l'aplicació
                menys l'alçada de l'asteroide
73          position.y =  new Random().nextInt(Settings.GAME_HEIGHT - (int) height)
                ;
74          setOrigin();
75      }
76
77
78      @Override
79      public void draw(Batch batch, float parentAlpha) {
80          super.draw(batch, parentAlpha);
81          batch.draw(AssetManager.asteroid[assetAsteroid], position.x, position.y
                , this.getOriginX(), this.getOriginY(), width, height, this.
                getScaleX(), this.getScaleY(), this.getRotation());
82      }
83
84      // Retorna true si hi ha col·lisió
85      public boolean collides(Spacecraft nau) {
86
87          if (position.x <= nau.getX() + nau.getWidth()) {
88              // Comprovem si han col·lisionat sempre que l'asteroide estigui a
                    la mateixa alçada que l'spacecraft
89              return (Intersector.overlaps(collisionCircle, nau.getCollisionRect
                    ()));
90          }
91          return false;
92      }
93  }
```

Podem crear i concatenar instruccions fent servir la classe `com.badlogic.gdx.scenes.scene2d.actions.Actions`, la documentació de la qual podeu trobar a https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/scenes/scene2d/actions/Actions.html. Això ens permetrà crear fàcilment accions.

La línia:

```
1   this.addAction(Actions.repeat(RepeatAction.FOREVER, Actions.rotateBy(-90f, 0.2f
        )));
```

és l'equivalent al codi anterior:

```
// Rotació
        RotateByAction rotateAction = new RotateByAction();
        rotateAction.setAmount(-90f);
        rotateAction.setDuration(0.2f);

        // Acció de repetició
        RepeatAction repeat = new RepeatAction();
        repeat.setAction(rotateAction);
        repeat.setCount(RepeatAction.FOREVER);

        this.addAction(repeat);
```

Podem concatenar les accions i fer coses com la següent:

```
spacecraft.addAction(Actions.repeat(RepeatAction.FOREVER, Actions.sequence(
     Actions.moveTo(0, 0), Actions.moveTo(50, 100, 2),Actions.moveTo(100, 50,
     5),Actions.moveTo(0, 0, 1), Actions.delay(2))));
```

Aquest codi faria de forma ininterrompuda (`RepeatAction.FOREVER`) la següent seqüència:

1. Posaria l'objecte en la posició (0,0).
2. Desplaçaria l'objecte durant 2 segons fins a la posició (50,100).
3. Després el desplaçaria durant 5 segons a la posició (100,50).
4. El tornaria a desplaçar a la posició (0,0) en 1 segon.
5. Esperaria 2 segons i tornaria a començar des del punt 1.

Cal consultar la documentació, hi ha molts mètodes útils que ens facilitaran la creació d'accions minimitzant la complexitat del codi.

2.3.10 Hit

Tal com tenim definit l'`InputProcessor` (`InputHandler`) no ens permet identificar sobre quins actors hem fet clic. Aquesta funcionalitat ens seria molt útil per tal d'afegir funcions als objectes o per crear elements amb els quals l'usuari pot interaccionar com per exemple botons (un botó seria un actor igual que la nau o els asteroides).

Per controlar aquesta possibilitat, haurem de tenir en compte que:

- Hem de definir els límits (*bounds* en anglès) de l'objecte cada cop que es mou.
- Hem de definir l'`actor` com a `touchable`.
- Quan fem clic sobre la pantalla, comprovarem si s'ha tocat algun element.

Per poder controlar quan es fa clic sobre la nau, anem al constructor d'`spacecraft` i afegim el codi:

```
// Per a la gestió de hit
        setBounds(position.x, position.y, width, height);
        setTouchable(Touchable.enabled);
```

I al final del mètode `act`:

```
setBounds(position.x, position.y, width, height);
```

Una vegada configurat l'objecte `spacecraft` cal anar a la classe `InputHandler` per controlar quan fa clic l'usuari. Creem les variables d'instància `private Vector2 stageCoord;` i `private Stage stage;`. Obtenim l'`stage` en el constructor amb `stage = screen.getStage();` i en el mètode `touchDown` afegim el següent codi:

```
stageCoord = stage.screenToStageCoordinates(new Vector2(screenX, screenY));
        Actor actorHit = stage.hit(stageCoord.x, stageCoord.y, true);
        if (actorHit != null)
            Gdx.app.log("HIT", actorHit.getName());
```

Convertim les coordenades de la pantalla a les de l'`stage` ja que per exemple a l'aplicació d'escriptori li hem duplicat les dimensions i l'aplicació de dispositius mòbils es redimensionarà per adaptar-se a la pantalla del dispositiu. Guardem l'`actor` que ha estat pressionat en les coordenades de la pantalla i si no és `null` mostrem un missatge de *log* amb el seu nom en cas de tenir-lo. El darrer paràmetre del mètode `hit` defineix si es respecta la `touchability` de l'objecte, és a dir, només mirarà actors que tinguin el `setTouchable` en `Touchable.enabled`.

Si executem l'aplicació i fem clic en la nau veurem missatges "HIT: spacecraft" en el *log* de l'Android Studio.

2.3.11 Configuració de l'aplicació d'Android

Si volem reduir el consum energètic generat pels sensors d'Android i millorar l'experiència de l'usuari, cal fer alguns canvis. La configuració de l'aplicació d'Android la farem des del fitxer del subprojecte *android/java/cat.xtec.ioc.android/AndroidLauncher*.

En la nostra aplicació utilitzarem ni l'acceleròmetre ni la brúixola així que afegirem entre `AndroidApplicationConfiguration config = new AndroidApplicationConfiguration();` i `initialize(new SpaceRace(), config);` les línies:

```
config.useAccelerometer = false;
config.useCompass = false;
```

D'aquesta manera evitarem el consum generat per aquests sensors.

Per millorar l'experiència de l'usuari en sistemes Android, configurarem l'aplicació per tal que mai s'apagui la pantalla i també ocultarem (en cas de tenir-los disponibles) els botons *software* de *home*, *back* i *recents* posant l'aplicació en el mode *immersive*.

El codi de la classe `AndroidLauncher` quedarà de la següent manera:

```
1  package cat.xtec.ioc.android;
2
3  import android.os.Bundle;
4
5  import com.badlogic.gdx.backends.android.AndroidApplication;
6  import com.badlogic.gdx.backends.android.AndroidApplicationConfiguration;
7  import cat.xtec.ioc.SpaceRace;
8
9  public class AndroidLauncher extends AndroidApplication {
10   @Override
11   protected void onCreate (Bundle savedInstanceState) {
12     super.onCreate(savedInstanceState);
13     AndroidApplicationConfiguration config = new
           AndroidApplicationConfiguration();
14     // Deshabilitem els sensors que hem de fer servir
15     config.useAccelerometer = false;
16     config.useCompass = false;
17     // Impedim que s'apagui la pantalla
18     config.useWakelock = true;
19     // Posem el mode immersive per ocultar botons software
20     config.useImmersiveMode = true;
21
22     // Apliquem la configuració
23     initialize(new SpaceRace(), config);
24   }
25 }
```