Játékfejlesztés

Hatékony kétdimenziós rajzolás Android platformon

Balogh Tamás

balogh.tamas@autsoft.hu

Tematika

- 2D Játékfejlesztés – Alapok
- Hatékony rajzolás – SurfaceView
- Kirajzoló ciklus és FPS Szabályozás
- Animációk és Spriteok
- Irányítás - Szenzorok használata

Felhasználói felület

- Játékalkalmazás
  - Egyedi játéknézet
- Saját felületi elemek
  - Játék objektumok
  - Háttér
  - Irányító felület (HUD)
- Egyedi rajzolási logika
  - Saját animációk
  - Fizikai szimuláció
- A rendszer által biztosított elemek hiánya
  - Esetleg menükben, beállításokban találkozunk velük.
  - Ritkán a HUD is ilyen

Egyedi felület

Előnyei

- Minden apró részlet általunk irányítható
- Erőforrás szabályozás is a mi kezünkben van

Hátrányai

- Nekünk kell implementálni -> több fejlesztés
- Nekünk kell az erőforrásokat kezelni
  - Felszabadítani
  - Állapotot menteni

Saját View készítése

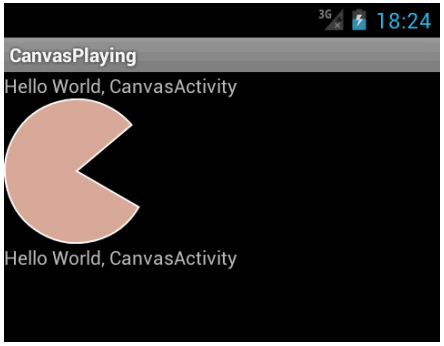
- Az osztályunk a View osztályból származik
  - class MyView: View
- Az onDraw függvény felüldefiniálása
  - override fun onDraw(c: Canvas)
- Rajzolás a Canvas-re
  - canvas.drawLine(...)
  - canvas.drawCicle(...)
  - canvas.drawRectangle(...)
  - canvas.drawBitmap(...) !!!!
- Egyedi rajzolás Paint objektumok segítségével

Demo

- CustomView készítése
- Kirajzolás CustomView-val – Statikus kép
- Kirajzolás CustomView-val – Mozgó kép

Saját View készítése

Előnyei



- Egyszerű
- A rendszer működésébe illeszkedik
  - Használható XML-ből, layoutba tehető stb…

Hátrányai

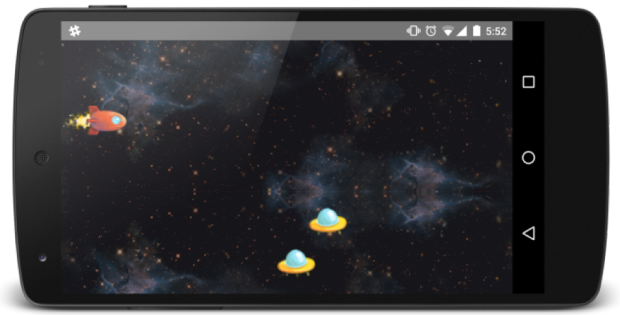
- A konkrét kirajzolás a főszálon fut (UI Thread)
  - Rajzolás alatt blokkol.
- Ideális 1-2 szövegre, vagy álló képekre, de egy összetettebb játékra nem

Saját nézet SurfaceView használatával

- Ez is egy View leszármazott
  - Layoutba tehető, stb…
- A rajzolást nem a View végzi
  - Csak egy felület amibe a rajzolt képet be lehet tölteni
- A rajzolás akár másik szálról is történhet
- Nem csak játékokhoz
  - Pl. camera alkalmazás
- SurfaceHolder objektum
  - Hozzáférést biztosít a nézethez egy másik szálról
  - lockCanvas és unlockCanvasAndPost metódusokkal zárolható a Canvas a rajzolás során

Saját nézet SurfaceView használatával

- A nézetünk a SurfaceView-ból származik
  - class GameView: SurfaceView
- A nézet változásának eseményeiről egy callback metódusban értesülhetünk
  - Létrehozás
  - Átmérezetés
  - Megszűnés
- Ezekkel az callback metódusokkal szabályozhatjuk a háttérben történő rajzolást.



Saját nézet SurfaceView használatával

```
holder.addCallback(object : SurfaceHolder.Callback {
    override fun surfaceCreated(holder: SurfaceHolder) {

    }

    override fun surfaceDestroyed(holder: SurfaceHolder) {
        //Stop rendering
    }

    override fun surfaceChanged(holder: SurfaceHolder, format: Int, width: Int,
height: Int) {
        //Resize and Start rendering
    }
})
```

Saját nézet SurfaceView használatával

- Használata ugyanaz, mint bármely más View-nak

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <hu.bme.aut.amorg.example.spaceshipgame.GameView
      android:id="@+id/gameView"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
</RelativeLayout>
```

Demo

- SurfaceView készítése
- Kirajzolás SurfaceView-val

Játékmodell

- Sima Java objektumok
  - Plain Old Java Object
- Célszerű ha van egy közös ősinterfészük ….

```
interface GameEntity{
    fun step()
    fun setSize(x: Int, y: Int)
    fun render(canvas: Canvas)
}
```

Játékmodell

- … amit minden entitás implementál

```
class Ship: GameEntity {

  override fun step() { //Recalculate state, position ...}

  override fun size(x: Int, y: Int) {//Set size ..}

  override fun render(canvas: Canvas) {
      // Preload bitmap in constructor
      canvas.drawBitmap(image, posx,posy, null)
  }

}
```

Játékmodell

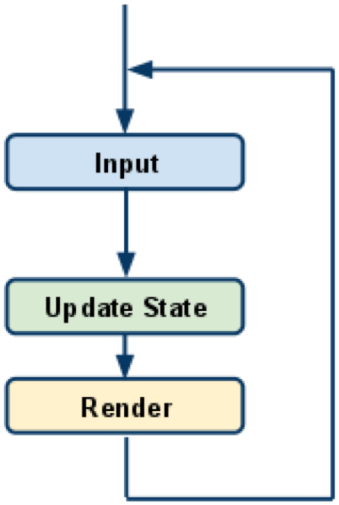
- A képi és más erőforrásokat célszerű a konstruktorban betölteni
  - Később ezt felhasználjuk minden rajzolási ciklusban
- A rajzolási ciklusban
  - A lehető legkevesebb műveletet végezzük
  - A lehető legkevesebb objektumot hozzuk létre

```
class Player(context: Context): Ship {
    //...
    val image = BitmapFactory.decodeResource(context.resources,
  R.drawable.ship)
}
```

Kirajzoló ciklus

Kirajzoló ciklus

- Külön szálon - Java Thread
- Ez a szál rajzolja ki a játékteret a SurfaceView-ba.
- Az egész kirajzolási ciklust irányítja
  - RenderLoop
- Minden egyes lépésben:
  - A játéktér állapotát újraszámolja
    - Lépteti az objektum állapotát
    - Mozgatás, ütközés detektálás
  - Zárolja a Canvas-t
  - Kirajzolja az objektumokat
  - Feloldja a Canvas zárolását
  - Vár (Fix FPS érdekében)



Kirajzoló ciklus

```
val renderStart = getTime()

// Játékmodell újraszámolása

var c = Canvas? = null
try {
   c = view.holder.lockCanvas()
   renderer.draw(c)
} finally {
   if (c != null) {
       view. holder.unlockCanvasAndPost(c)
   }
}
```

Kirajzoló ciklus

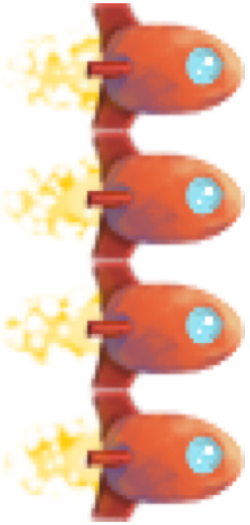
```
val renderEnd = getTime()

val sleepTime = timeBetweenFrames - (renderEnd - renderStart);

if (sleepTime > 4) {
   sleep(sleepTime)
} else {
   sleep (5)
}
```

Demo

- Entitások kirajzolása
- Entitások mozgatása

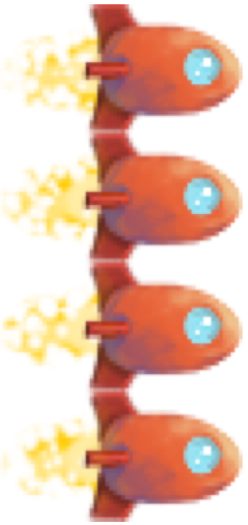
Animáció

- Animáció Sprite-ok segítségével
- Különböző állapotokban más-más képet jelenítünk meg
- Különböző állapotokat egy-egy képrészlet reprezentál
  - Járás, Robbanás …
- Minden állapot 1 képen van rajta
- Az összes állapot egy Bitmapben van betöltve a memóriába
- Az adott állapot kirajzolásához egy eltolással maszkoljuk ki a képet



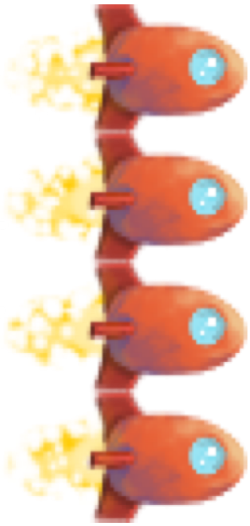
Animáció

```
class Player(context: Context): Ship{
   val SPRITE_HORIZONTAL = 1
   val SPRITE_VERTICAL = 4

   init {
     image = BitmapFactory.decodeResource(context.getResources(), R.drawable.ship)
   }

   override fun step() {
     state++
   }

  fun setSpriteSizes() {
       spriteWidth = image.width/ SPRITE_HORIZONTAL
       spriteHeight = image.height / SPRITE_VERTICAL
   }

}
```



Animáció

```
override fun render(canvas: Canvas) {
    setSpriteSizes()
    val statePos = state/5

    val x = 0
    val y = spriteHeight * statePos

    val src = Rect(x, y, x + spriteWidth, y + spriteHeight)
    val dst = Rect(posX, posY, posX + spriteWidth * 4, posY + spriteHeight * 4)

    canvas.drawBitmap(image, src, dst, null)
}
```



Demo

- Animációk Sprite-ok segítségével.

Demo

- Sensor demo

Köszönöm a figyelmet!

balogh.tamas@autsoft.hu

AutSoft