Departamento de Informática | Universidad de Valladolid

Sistemas Móviles

El juego de la vida

Félix Prieto

Curso 2013/14

El juego de la vida

- Diseñado por John Horton Conway en 1970
- Un tablero bidimensional poblado de «células» vivas y muertas
- La población cambia de generación en generación:
  - Una célula muerta revive si está rodeada por exactamente tres células vivas
  - Una célula viva muere a no ser que esté rodeada de dos o tres células vivas
- Se han documentado diversos patrones que dan lugar a comportamientos curiosos
- Implementación en java disponible en linea `http://www.bitstorm.org/gameoflife/`

Objetivos de la aplicación

- Elaboración de una primera aplicación funcional
- Elaboración de una interfaz basada en un gráfico 2D
- Utilización de clases externas al proyecto
- Aprendizaje de la importancia de la gestión eficiente de recursos
- Introducción muy elemental a la utilización de hilos de ejecución

Requisitos básicos (muy informales

- La aplicación:
  - Mostrará la evolución de un universo prefijado
  - Permitirá guardar el estado de una simulación para recuperarlo posteriormente
  - Permitirá guardar y recuperar un universo desde la memoria externa
  - Permitirá editar un universo en cualquier momento
- Algunos de los requisitos quedarán seguramente como ejercicio en clase

Utilizar clases de otro proyecto

- Podemos utilizar clases Java de otros proyectos
- Las mismas clases pueden ser usadas en muchos proyectos
- Hay que añadir la información a las propiedades de nuestro proyecto
- Project-> Properties->Java Build Path -> Projects -> Add
- También se pueden usar ficheros jar
- Project-> Properties->Java Build Path -> Libraries -> Add
- Un proyecto se puede exportar a formato jar desde el propio Eclipse

Algunos matices

- A partir de la versión 17 del SDK Google recomienda otra forma de importación:
  - Crear una carpeta `libs` en la raíz del proyecto
  - Copiar a ella todos los `.jar` deseados
- «Probablemente» sólo se pueden importar paquetes `jar` de la versión 1.6 de Java
- El problema está en la traducción al formato de la máquina virtual de Android

Hilos de ejecución

- Un hilo es una tarea que puede ejecutarse en paralelo con otras
- Varios hilos comparten los mismos recursos
- En Java, podemos utilizar hilos extendiendo la clase `Thread` y redefiniendo el método `run()`
- Arrancamos el hilo mediante el método `start()`
- Paramos el hilo forzando la detención del método `run()`
- También se pueden usar hilos implementando la interfaz `Runnable`
- El estudio de los hilos de ejecución excede los objetivos de esta asignatura.

Definición de un hilo

```
class Hilo extends Thread {
    private SurfaceHolder surfaceHolder;
    private VistaConHilosVida vista;
    private boolean funcionando;

    public Hilo(SurfaceHolder s,
        VistaConHilosVida v) {
      surfaceHolder = s; vista = v;
    }
    public void termina() {
      funcionando = false;
    }
    @Override
    public void run() {
      Canvas c; funcionando = true;
      while (funcionando) {
        c = surfaceHolder.lockCanvas();
        vista.dibuja(c);
        surfaceHolder.unlockCanvasAndPost(c);
      }
    }
  }
```

Parar un hilo

```
public void terminaHilo() {
  hilo.termina();
  boolean falla = true;
  int i = 1;
  while (falla) {
    try {
      Log.d(TAG, TAG + ":␣Join␣intento␣" + i);
      hilo.join();
      falla = false;
    } catch (InterruptedException e) {
      i++;
    }
  }
}
```

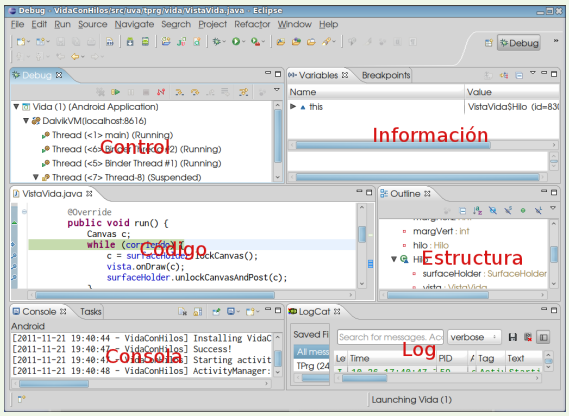

Uso de un hilo

```
// Lanzar el hilo
hilo = new Hilo(holder, this);
hilo.start();
//...
// Detener el hilo
terminaHilo();
```


Uso del depurador

- Eclipse dispone de un depurador de código
- Puede usarse incluso con un dispositivo externo
  - Hay que añadir `android:debuggable="true"` en el elemento `<application>` dentro de `AndroidManifest.xml`
- Habitualmente se usa desde una perspectiva específica
- Se añade algún «BreakPoint»
- Se ejecuta mediante el icono correspondiente o mediante Run -> Debug
- La ejecución se realiza normalmente hasta llegar al «BreakPoint»
- Se pueden añadir nuevos puntos durante la depuración

La perspectiva «debugger»



Uso del depurador

- Podemos analizar el valor de las variables durante la ejecución
- Podemos añadir puntos de parada condicionales,...
- Controlamos la ejecución a partir de la primera parada
  - F5 (Step Into) Continúa la ejecución parando en los métodos a los que llamamos
  - F6 (Step Over) Continúa la ejecución sin parar en los métodos a los que llamamos
  - F7 (Step Return) Continúa la ejecución parando en el código del método que nos invoca
  - F8 (Resume) Continúa la ejecución hasta el siguiente «BreakPoint»
- Sólo muestra el código disponible desde eclipse
- «Android Source Plugin» permite ver el código del framework (http://code.google.com/p/adt-addons/)

Implementación de la interfaz

- Para mostrar las células y su evolución podemos utilizar varias estrategias:
  - Una interfaz basada en botones o etiquetas: Presenta problemas con universos de tamaño moderado o grande
  - Una interfaz dibujada en un descendiente de `View`: La interfaz se desenvuelve con menos soltura
  - Una interfaz dibujada en un descendiente de `SurfaceView`: Presenta la complicación extra de gestionar hilos
- Sólo comentaremos las peculiaridades de las dos últimas alternativas
- En ambos casos, nuestra clase será cargada en una `Activity` mediante `setContentView(vista)`

Dibujar en una vista

- Dibujamos sobre un objeto `Canvas` mediante `drawPath(path,paint)`
- El objeto `Path` representa el elemento que será dibujado
  - Creamos un nuevo objeto `Path`
  - Le añadimos elementos geométricos: círculos (`addCircle`), curvas cúbicas de Bezier (`cubicTo`),...
- El objeto `Paint` representa el estilo y color con que será dibujado
  - Creamos un nuevo objeto `Paint`
  - Le añadimos color con `setColor`
  - Podemos añadir sombras (`setShadowLayer`) u otros efectos.

Dónde escribir los comandos de dibujo

- Los comandos anteriores son invocados por el método `onDraw` de la vista
- Quien nos invoca desde el código del framework nos pasa el objeto `Canvas` como argumento
- La invocación a `onDraw` se realiza cuando el framework «lo considera necesario»
- Podemos forzar la llamada al método «invalidando» todo o parte de la vista mediante una llamada a `invalidate`
- Para conseguir un efecto de «animación» recalculamos el dibujo e invalidamos la vista al final del propio método `onDraw`.

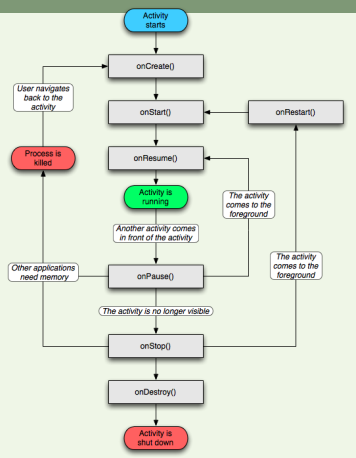
Uso de colores

- Implementados mediante un entero que guarda valores ARGB
- La clase `Color` permite crearlos de forma cómoda
  - `paint.setColor(Color.RED);`
  - `paint.setColor(Color.argb(150, 255, 0, 0));`
  - `paint.setColor(0x96ff0000);`
  - `paint.setColor(getResources().getColor(R.color.miColorRojo));`
- Lo ideal es utilizar recursos de color

Cambio de tamaño de la vista

- En cualquier momento podemos determinar el tamaño de la vista
  - `getWidth()` y `getHeight()`
- Durante la creación de la vista el tamaño puede no ser el esperado
- Podemos redefinir el método `onSizeChanged()` que será invocado cuando cambie el tamaño
- En todo caso hay que comprobar el tamaño para «dibujar» la vista

Ciclo de vida de las actividades



- Debemos prevenir la pérdida del foco por parte de nuestra aplicación:
  - `onPause` debe guardar el estado actual de la actividad
  - `onResume` debe recuperar el estado de la actividad
- En general, necesitaremos utilizar ficheros

Nociones básicas sobre ficheros en Android

- Cada aplicación se corresponde con un usuario UNIX
  - Su nombre es el nombre del paquete
- Cada aplicación puede usar libremente su directorio *HOME*
  - El *HOME* de una aplicación es `/data/data/<paquete>`
  - Para evitar problemas con versiones posteriores de Android utilizar `getFilesDir()` de la clase `Context`
- La clase `Context` proporciona algunos servicios de manejo de ficheros (`deleteFile`, `fileList`,...)
- En general, el manejo de ficheros utiliza las técnicas habituales en java.

Escribiendo en un fichero

```
@Override
protected void onPause() {
  super.onPause();
  Log.d(TAG, TAG + ": Escritura en fichero");
  BufferedWriter fichero;
  try {
    fichero = new BufferedWriter(new FileWriter(
                getFilesDir()+"/prueba.txt"));
    fichero.write("Escribo en un fichero");
    fichero.newLine();
    fichero.close();
  } catch (Exception e) {
    Log.d(TAG, TAG + ": Error al escribir");
    e.printStackTrace();
  }
}
```


Leyendo de un fichero

```
@Override
protected void onResume() {
  super.onResume();
  Log.d(TAG, TAG + ": Lectura de un fichero");
  BufferedReader fichero;
  try {
    fichero = new BufferedReader(new FileReader(
                getFilesDir()+"/prueba.txt"));
    String linea;
    while ((linea = fichero.readLine()) != null)
      Log.d(TAG, TAG + ": Última lectura ---->"
            + linea);
  } catch (Exception e) {
    Log.d(TAG, TAG + ": Error al leer ");
    e.printStackTrace();
  }
}
```


Ficheros en la SD

- Las aplicaciones también pueden usar la «SD»
  - Está montada en `/mnt/sdcard`
  - Es preferible utilizar indirecciones para el directorio:
    - `getExternalFilesDir(null)` para el directorio de la aplicación en la SD
    - `getExternalFilesDir(DIRECTORY_ALARMS)` para el directorio de alarmas en la SD
    - `Environment.getExternalStorageDirectory()` para los directorios de uso general de la SD
- Para escribir en la SD es preciso disponer de permisos especiales declarados en `AndroidManifest.xml`
- Las funciones usadas para abrir, cerrar, leer o escribir ficheros son las mismas

Extender `SurfaceView`

- Extendemos `SurfaceView` e implementamos `SurfaceHolder.Callback`
- Esta interfaz requiere que implementemos tres métodos para reaccionar a los cambios de la vista: `surfaceCreated`,`surfaceDestroyed` y `surfaceChanged`
- Cuando se inicie la simulación lanzaremos un hilo que redibuje continuamente la vista
- Para parar la simulación destruiremos el hilo
- Los comandos de dibujo son iguales a los de la versión anterior, pero ahora no «invalidamos» la vista.
- Es preciso redibujar el fondo al principio de cada actualización

El selector de ficheros

- El framework de Android no dispone de un diálogo para seleccionar ficheros
- Si realmente lo necesitamos podemos:
  - Implementarlo desde cero
  - Utilizar una biblioteca externa
  - Utilizar una aplicación externa
- Quizás en realidad no necesitamos un selector de ficheros.

Utilizar una biblioteca externa

- `http://code.google.com/p/android-file-dialog/`
- Lanzamos una actividad externa para seleccionar el nombre del fichero
- Utilizamos `startActivityForResult` para poder recuperar el resultado
- La actividad llamada utiliza `setResult(codigo, intent)` para devolver un resultado
- Al terminar la actividad el sistema llama a `onActivityResult()`
  - `requestCode` es el código de la llamada
  - `resultCode` es el código del resultado
  - `data` encapsula los datos enviados mediante un `Intent`

Utilizar biblioteca externa Android

- Imprescindible si se utilizan conceptos del Framework (Recursos, actividades,...)
- Hay que declarar la biblioteca en Properties -> Android -> Library
- Todos los proyectos usados tienen que estar abiertos para compilar

Utilizar una aplicación externa

- `http://www.blackmoonit.com/` ofrece un administrador de archivos «bien documentado»
- Lanzaremos un `intent` implícito mediante `startActivityForResult`
- Podemos detectar desde el código si existe alguna aplicación capaz de aceptar el `intent`
  - Utilizamos `PackageManager`
- Podemos ofrecer la instalación de la aplicación si es necesario
  - En el emulador no funciona si no hemos instalado Android Market

Ofrecer nuestra aplicación

- Podemos «ofrecer» alguna de nuestras actividades para realizar un trabajo
- Declaramos un `intent-filter` asociado esa actividad
  - Qué tareas podemos hacer
  - Qué tipo de datos podemos procesar
  - Más detalles en `http://developer.android.com/guide/components/intents-filters.html`
- Si se ofertan varias aplicaciones el usuario puede seleccionar una de ellas
- `PackageManager` se ocupa de esas «gestiones»
- Si no existe ninguna aplicación disponible se genera una excepción

Ofrecer nuestra aplicación:intent-filter

- El `intent-filter` aparece en el fichero `AndroidManifest.xml`
- Cualquier aplicación tiene al menos un `intent-filter` asociado a su actividad principal
- Podemos añadir `intent-filter` asociados a cualquier actividad
- El filtro ofrece una categoría, acción, esquema de datos,...

Ofrecer nuestra aplicación: Procesar el intent

- Una actividad puede ofrecer varios filtros
- Una actividad puede ser iniciada de varios modos
- Podemos acceder al objeto `Intent` que nos activó mediante `getIntent()`
- Discriminamos las formas de activación en función de la información de nuestro objeto `Intent`
- Antes del final de la actividad debemos invocar al método `setResult` para devolver un resultado
- Mediante `setResult` podemos fijar un código para la respuesta y un `Intent` que será recibido por la otra actividad