


Programación de Dispositivos Móviles




Sesión 14:
Juegos

Java y Dispositivos Móviles

© 2007-2009 Depto. Ciencia Computación e IA

Juegos-1

Índice




- Juegos para móviles
- Desarrollo de juegos
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

Java y Dispositivos Móviles

© 2007-2009 Depto. Ciencia Computación e IA

Juegos-2

Juegos



- Juegos para móviles
- Desarrollo de juegos
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

Java y Dispositivos Móviles

© 2007-2009 Depto. Ciencia Computación e IA

Juegos-3

Juegos para móviles



- Los juegos Java han tenido un gran éxito
 - Gran parte de los usuarios de móviles están interesados en este tipo de aplicaciones de ocio
 - Java nos permite portar fácilmente los juegos a distintos modelos de móviles con poco esfuerzo
- Es el tipo de aplicación MIDP más difundida



Características de los juegos para móviles



- Normalmente el usuario utiliza el móvil para pasar el rato mientras hace tiempo
 - Cuando está haciendo cola
 - Cuando viaja en autobús
 - Etc
- Por lo tanto estos juegos deberán
 - No requerir apenas aprendizaje por parte del usuario
 - Permitir ser pausados
 - El usuario puede ser interrumpido en cualquier momento
 - Permitir guardar nuestro progreso
 - Cuando tengamos que dejar el juego, no perder nuestros avances

Características de los dispositivos



- Escasa memoria
 - No crear más objetos de los necesarios
 - No cargar excesivo número de recursos, ni imágenes complejas
- CPU lenta
 - Optimizar el código
- Pantalla reducida
 - Los gráficos deben ser suficientemente grandes
- Almacenamiento limitado
 - Almacenar la información mínima al guardar la partida
 - Codificar la información de forma compacta
- Poco ancho de banda
 - Dificultad para implementar juegos de acción en red
- Teclado pequeño
 - El manejo debe ser sencillo
- Posibles interrupciones
 - Permitir modo de pausa

Juegos



- Juegos para móviles
- **Desarrollo de juegos**
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

API de bajo nivel



- **Los juegos deben resultar atractivos a los usuarios**
 - Deberán tener gráficos personalizados e innovadores
 - Deberemos dar una respuesta rápida al control del usuario
- **Utilizaremos por lo tanto la API de bajo nivel**
 - Nos permite dibujar gráficos libremente
 - Tenemos acceso completo a los eventos del teclado
- **En MIDP 2.0 se incluye una librería para desarrollo de juegos**
 - Incorpora clases que nos facilitarán la creación de juegos

`javax.microedition.lcui.game`

Aplicación conducida por los datos



- **El motor del juego debe ser lo más genérico posible**
- **Debemos llevar toda la información posible a la capa de datos**
 - El juego normalmente consistirá en varios niveles
 - La mecánica del juego será prácticamente la misma en todos ellos
 - Será conveniente llevar la definición de los niveles a la capa de datos
 - Almacenaremos esta información en un fichero
 - Simplemente editando este fichero, podremos añadir o modificar los niveles del juego

Ejemplo de codificación de niveles



- Vamos a ver como ejemplo un clon del *Frogger*
- El fichero con los datos de niveles estará codificado de la siguiente forma:

```
<int> Numero de fases
Para cada fase
  <UTF> Titulo
  <byte> Número de carriles
Para cada carril
  <byte> Velocidad
  <short> Separación
  <byte> Tipo de coche
```

- Podemos utilizar un objeto `DataInputStream` para deserializar esta información

¿Y eso no puede hacerlo otro?



- Para realizar tareas anodinas y repetitivas están las máquinas
 - J2ME-Polish incluye un editor de ficheros binarios genérico

Name	Count	Type	Data
NumFases	4	int	4
Titulo	4	UTF-String	Fase 1
NumCarriles	4	byte	2
Carriles	NumCarriles	TrackData	[3, 100, 2] [2, -250, 0] [4, 200, 1]
Titulo	4	UTF-String	Fase 2
NumCarriles2	4	byte	2
Carriles2	NumCarriles2	TrackData	[4, 150, 2] [1, -100, 1] [5, 150, 0] [2, 200, 1] [5, 100, 1]
Titulo	4	UTF-String	Fase 3
NumCarriles3	4	byte	2
Carriles3	NumCarriles3	TrackData	[4, 150, 0] [5, -200, 1] [5, 200, 0] [2, 100, 2]
Titulo	4	UTF-String	Fase 4
NumCarriles4	4	byte	2
Carriles4	NumCarriles4	TrackData	[3, 150, 2] [5, -250, 1] [2, 150, 0] [4, 100, 1] [2, 75, 2]

www.j2mepolish.org

- Además incluye otras herramientas y librerías útiles para el desarrollo de juegos (editor de fuentes, componentes propios, etc) y para hacerlos independientes del modelo de dispositivo

Gestión de recursos



- Conviene centralizar la gestión de los recursos
 - Crear un clase que gestione la carga de estos recursos
- Cargar todos los recursos necesarios durante la inicialización
 - En dispositivos con poca memoria podríamos cargar sólo los recursos de la fase actual
- No tener los recursos dispersos por el código de la aplicación
 - Evitamos que se carguen varias veces accidentalmente
 - Los recursos los carga el gestor de recursos y todos los demás componentes de la aplicación los obtendrán de éste
 - Sabremos qué recursos está utilizando la aplicación simplemente consultando el código del gestor

Gestor de recursos



```
public class Resources {
    public static final int IMG_TIT_TITULO = 0;
    public static final int IMG_SPR_CROC = 1;
    public static final int IMG_BG_STAGE_1 = 2;

    public static Image[] img;

    private static String[] imgNames = {
        "/title.png", "/sprite.png", "/stage01.png" };

    private static void loadCommonImages()
        throws IOException {
        img = new Image[imgNames.length];
        for (int i = 0; i < imgNames.length; i++) {
            img[i] = Image.createImage(imgNames[i]);
        }
    }
}
```

Tipos de recursos



- En el gestor de recursos podemos añadir recursos como
 - Imágenes
 - Sonidos
 - Datos
 - Etc
- Podremos acceder a los datos de las fases a través de este gestor
 - Los datos habrán sido leídos del fichero de datos de fases durante la inicialización
 - Debemos haber creado las estructuras de datos adecuadas en Java para encapsular esta información

Portabilidad



- Centralizar toda la información dependiente del dispositivo en una misma clase
 - Dimensiones de la pantalla
 - Dimensiones de los objetos
 - Posiciones de los objetos
 - Etc
- Podemos añadir estos datos como constantes en una misma clase

```
public class CommonData {
    public static final int NUM_LIVES = 3;
    public final static int SCREEN_WIDTH = 176;
    public final static int SCREEN_HEIGHT = 208;
    public final static int SPRITE_WIDTH = 16;
    public final static int SPRITE_HEIGHT = 22;
    ...
}
```

Optimización



- El juego deberá funcionar de forma fluida para que sea jugable
 - Deberemos optimizarlo
 - Primero hacer una implementación clara, después optimizar
- Identificar que operación consume más tiempo
 - Gráficos
 - Volcar sólo la parte de la pantalla que haya cambiado
 - Lógica
 - No crear más objetos que los necesarios, reutilizar cuando sea posible
 - Permitir que se desechen los objetos que no se utilicen, poniendo sus referencias a null

Juegos

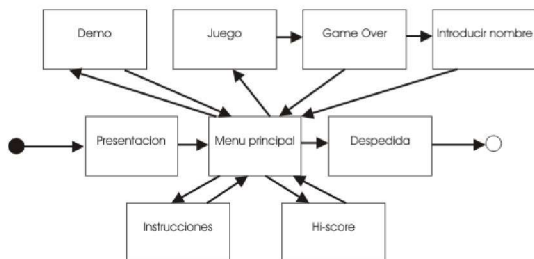


- Juegos para móviles
- Desarrollo de juegos
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

Pantallas



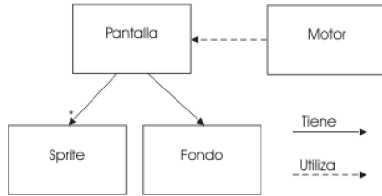
- Un juego típico suele constar de las siguientes pantallas



Componentes



- En la pantalla de juego, podemos distinguir los siguientes componentes:



Motor del juego



- Vamos a ver cómo implementar la pantalla en la que se desarrolla el juego
- Utilizaremos lo que se conoce como ciclo del juego
 - Bucle infinito
 - En cada iteración:
 - Lee la entrada
 - Actualiza la escena según la entrada e interacciones entre objetos de la misma
 - Redibuja los gráficos de la escena
 - Duerme durante un tiempo para que los ciclos tengan una duración uniforme
 - En la clase `GameCanvas` de MIDP 2.0 encontraremos facilidades para la creación de este ciclo

Ciclo del juego



- Crearemos el ciclo en una clase que herede de `GameCanvas`
 - El ciclo se ejecutará dentro de un hilo
 - Podemos poner en marcha el hilo en el evento `showNotify`

```
Graphics g = getGraphics();
long t1, t2, td;
while(true) {
    t1 = System.currentTimeMillis();
    int keyState = getKeyStates();
    tick(keyState);
    render(g);
    flushGraphics();
    t2 = System.currentTimeMillis();
    td = t2 - t1; td = td < CICLO ? td : CICLO;
    try {
        Thread.sleep(CICLO - td);
    } catch (InterruptedException e) { }
}
```

Máquina de estados



- Durante el desarrollo del juego pasaremos por diferentes estados

- En cada uno se permitirán realizar determinadas acciones y se mostrarán determinados gráficos
- Según el estado en el que nos encontremos el ciclo del juego realizará tareas distintas



Juegos



- Juegos para móviles
- Desarrollo de juegos
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

Entrada en MIDP 1.0



- En MIDP 1.0 podemos utilizar las acciones de juego para responder a los eventos del teclado
- Para conocer las teclas que se presionen deberemos capturar el evento de pulsación del teclado

```
public void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);
    if (action == LEFT) {
        moverIzquierda();
    } else if (action == RIGHT) {
        moverDerecha();
    } else if (action == FIRE) {
        disparar();
    }
}
```

Entrada en MIDP 2.0



- No hará falta capturar los eventos del teclado
- El método `getKeyStates` nos dirá las teclas pulsadas actualmente
 - Es más apropiado para implementar el ciclo del juego
 - La entrada del usuario se leerá de forma síncrona con el ciclo del juego
- Nos devolverá un entero en el que cada bit codifica la pulsación de una tecla

```
int keyState = getKeyStates();
```

- Podremos saber si una determinada tecla está pulsada utilizando una máscara como la siguiente

```
if ((keyState & LEFT_PRESSED) != 0) {  
    moverIzquierda();  
}
```

Juegos



- Juegos para móviles
- Desarrollo de juegos
- Motor del juego
- Entrada de usuario
- Componentes de la pantalla

Sprites



- Objetos que aparecen en la escena
 - Se mueven o podemos interactuar con ellos de alguna forma



- Lo creamos a partir de la imagen con el mosaico de *frames*

```
Sprite personaje = new Sprite(imagen,  
    ancho_frame, alto_frame);
```

- Podremos animar este *sprite* por la pantalla

Animación de los sprites



▪ Podemos mover el *sprite* por la pantalla

- Situar en una posición absoluta

```
personaje.setPosition(x, y);
```

- Desplazar respecto la posición actual

```
personaje.move(dx, dy);
```

▪ Podemos cambiar el *frame* del *sprite* para animarlo

```
personaje.setFrame(indice);
```

- Podemos establecer una secuencia de frames para la animación

```
this.setFrameSequence(new int[]{ 4, 5, 6});
```

- Para cambiar al siguiente frame de la secuencia actual llamaremos a

```
this.nextFrame();
```

- Para volver a disponer de la secuencia completa llamaremos a

```
personaje.setFrameSequence(null);
```

Colisiones de los sprites



▪ Muchas veces necesitaremos conocer cuando dos *sprites* “chocan” entre ellos. Por ejemplo

- Cuando el *sprite* de un enemigo toque a nuestro personaje, perderemos una vida
- Cuando una de nuestras balas impacten contra un enemigo, el enemigo morirá

▪ Esta información la obtendremos mediante cálculo de colisiones

```
personaje.collidesWith(enemigo, false);
```

- El rectángulo (*bounding box*) que se utilizará para calcular las colisiones tendrá como tamaño el tamaño de los *frames* de la imagen
- Podremos cambiar este rectángulo con

```
this.defineCollisionRectangle(x, y, ancho, alto);
```

Fondo

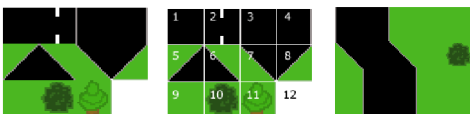


▪ Los *sprites* se moverán sobre un escenario de fondo

- El escenario muchas veces es más grande que la pantalla
- No es conveniente crear una imagen con todo el fondo ya que será demasiado grande

▪ Podemos construir el fondo como un mosaico

- Necesitaremos una imagen con los elementos básicos
- Compondremos el fondo a partir de estos elementos



Mosaico de fondo



- MIDP 2.0 nos proporciona la clase `TiledLayer` con la que crear el mosaico de fondo

```
TiledLayer fondo = new TiledLayer(columnas, filas,
                                  imagen, ancho, alto);
```

- Donde
 - `(columnas x filas)` serán las dimensiones del mosaico en número de celdas
 - `(ancho x alto)` serán las dimensiones en píxeles de cada elemento del mosaico
- Por lo tanto, el fondo generado tendrá unas dimensiones en píxeles de `(columnas*ancho) x (filas*alto)`
- Para fijar el tipo de elemento de una celda utilizamos

```
fondo.setCell(columna, fila, indice);
```

- Con el índice indicamos el elemento que se mostrará en dicha celda
 - Los elementos de la imagen se empezarán a numerar a partir de 1
 - Con 0 especificamos que la deje vacía (con el color del fondo)
 - Utilizaremos valores negativos para crear animaciones

Pantalla



- En la pantalla deberemos mostrar todos los elementos de la escena
 - Fondo
 - Sprites
- Podemos considerar que todos estos elementos son capas
 - Tanto `Sprite` como `TiledLayer` derivan de la clase `Layer`
 - Según el orden en el que se dibujen estas capas veremos que determinados objetos taparán a otros
- Podemos utilizar la clase `LayerManager` para crear esta estructura de capas

```
LayerManager escena = new LayerManager();
escena.append(personaje);
escena.append(enemigo);
escena.append(fondo);
```

- La primera capa que añadamos será la que quede más cerca del observador, y tapará a las demás capas cuando esté delante de ellas

Volcado de los gráficos



- Si tenemos una escena de gran tamaño, podemos hacer que sólo se muestre un determinado recuadro

```
escena.setViewWindow(x, y, ancho, alto);
```

- Esto nos permitirá implementar fácilmente *scroll* en el fondo
 - Haremos que el visor se vaya desplazando por la escena conforme el personaje se mueve
- Para volcar los gráficos en la pantalla utilizaremos

```
escena.paint(g, x, y);
```
