INTERFACES DE USUARIO CON LIBGDX

Autores

- Carlos Aguilar de la Morena.
- Ignacio Huesa Cerdán.
- Carlos López Rodríguez.
- Andrés Tamayo Jímenez.

Descarga Del Proyecto

Descargar código fuente del proyecto.

Tecnologías Usadas

- LibGDX:
  - Framework de desarrollo de videojuegos en Java que provee una API unificada que funciona en todas las plataformas soportadas (Windows, Linux, MacOS X, Android, BlackBerry, iOS, Java Applet, JavaScript/WebGl…).

- gdx-lml:
  - LibGDX Markup Language es un framework que permite parsear templates similares a HTML con macros inspiradas en FreeMarker a actores de `Scene2D`. Debido a que la creación de interfaces en Java puede ser una tarea tediosa e ininteligible gracias a la verbosidad del lenguaje, LML puede ser una alternativa bastante útil. Especialmente gracias a que los templates de LML se pueden modificar y recargar sin tener que recompilar la aplicación completa. Contiene soporte extra para gestionar preferencias, assets e internacionalización.

- VisUI:
  - VisUI permite la creación de interfaces de usuario con un diseño crafteado al detalle usando widgets de `Scene2d.ui` para proveer algunas funcionalidades extra como bordes indicadores de foco, cambios de fondos basados en eventos, etc.

- lml-vis:
  - Permite parsear templates de LML a widgets de VisUI, en lugar de a los estándar de `Scene2D` y extiene la sintaxis con modos de contruir los nuevos actores. Incluso si quieres tener un aspecto personalizado en tu aplicación, considera usar esta librería por los widgets mejorados.

Descripción De Las Tecnologías

LibGDX pone a disposición de los desarrolladores un conjunto de componentes para implementar interfaces de usuario: Scene2d.ui.

En conjunto con este paquete de componentes, también pone a nuestra disposición un lenguaje de remarcado para el desarrollo de interfaces gráficas: gdx-lml. LML está basado en XML, usando etiquetas para declarar componentes, a las cuales se les puede añadir atributos con los que configurar dichos componentes.

Si bien LibGDX también dispone de editores gráficos como Overlap2D o VisEditor, nos hemos decantado por LML debido a la escalabilidad y libertad de configuración que nos proporciona.

A parte de los componentes que LibGDX trae por defecto, VisUI redefine y rediseña gran parte de estos componentes, además de añadir algunos nuevos. VisUI contiene una redefinición de LML, lml-vis, con la que poder editar interfaces con los componentes de ésta usando LML.

Elementos Usados

VisWindow

Este componente es una tabla con una barra de título encima de los componentes. Opcionalmente puede actuar como un diálogo modal, previniendo eventos de ratón (o táctiles) para los widgets que haya debajo. `VisWindow` define un fondo que puede ser un mapa de bits y una fuente y su respectivo color para el título.

VisWindow.lml

```
<?xml version="1.0"?>
<!DOCTYPE viswindow SYSTEM "../../lml.dtd">

<viswindow onecolumn="true" align="left" fillparent="true" padtop="8">
    <!-- CONTENT -->
</viswindow>
```

VisTable

La clase `VisTable` redimensiona y posiciona sus widgets hijos usando una tabla lógica, similar a las tablas de HTML. Se supone un uso extensivo de las en `scene2d.ui` para organizar los widgets, debido a su facilidad de uso y a la potencia en comparación con redimensionar y posicionar los widgets manualmente. Los layouts basados en tablas no se apoyan sobre un posicionamiento absoluto y, por lo tanto, se ajustan a diferentes tamaños de widgets y resoluciones de pantalla.

VisTable.lml

```
<vistable row="true" tablealign="left" fillparent="true" filly="true">
    <:row padtop="10" padbottom="10">
        <!-- CONTENT -->
    </:row>
    <:row>
        <!-- CONTENT -->
    </:row>
</vistable>
```

VisTextField

Se trata de un campo de entrada de texto de una única línea. Contiene definiciones de `Drawables` para el fondo, un cursor de texto, un estilo de selección de texto, una fuente y su respectivo color para cuando se introduce texto, y una fuente y su respectivo color para el mensaje mostrado cuando el campo está vacío. Se puede activar el modo password, con el cuál se mostrarán asteriscos en lugar del texto introducido.

VisTextField.lml

```
1 <vistextfield id="path" colspan="2" align="left" expandx="true" fillx=
```

VisSplitPane

Contiene dos widgets y está dividido en dos, ya sea horizontal o verticalmente. El usuario podría redimensionar los widgets con un divisor arrastrable. Los widgets hijos siempre se redimensionan para llenar su respectiva mitad del `VisSplitPane` . `VisSplitPane` define un `Drawable` para el divisor arrastrable.

VisSplitPane.lml

```
1 <vissplitpane align="left" expandx="true" fillx="true" height="650" sp
2     <!-- CONTENT -->
3 </vissplitpane>
```

VisScrollPane

El `VisScrollPane` navega por el widget hijo usando barras de navegación y/o el ratón o arrastre táctil. La navegación se activa automáticamente cuando el widget es más grande que el `VisScrollPane` . Si el widget es más pequeño que el `VisScrollPane` en una dirección, se redimensiona para que adopte el tamaño de éste en dicha dirección. `VisScrollPane` tiene muchos ajustes para indicar si los toques controlan la navegación y cómo, desvanecimiento de las barras de navegación, etc. `VisScrollPane` define `Drawables` para el fondo y las barras de navegación horizontal y vertical. Se el control táctil está habilitado (por defecto), todos los `Drawables` son opcionales.

VisScrollPane.lml

```
1 <visscrollpane>
2     <!-- CONTENT -->
```

```
3 | </visscrollpane>
```

VisTree

Muestra una jerarquía de nodos. Cada nodo puede tener nodos hijos que pueden ser expandidos o contraídos. Cada nodo contiene un actor, permitiendo flexibilidad completa sobre cómo cada item se muestra. `VisTree` define `Drawables` para los iconos de expansión y contracción al lado de los actores de cada nodo.

VisTree.lml

```
1 | <vistree id="tree" padding="1"></vistree>
```

GridGroup

Se trata de un widget que ordena múltiples widgets en forma de rejilla. Cada elemento del casillero puede contener un actor, permitiendo flexibilidad completa sobre cómo cada item se muestra. `GridGroup` define el tamaño y el espaciado de los items.

GridGroup.lml

```
1 | <gridgroup id="board"></gridgroup>
```

Inicialización Del Parser De LML

Antes de empezar a renderizar vistas definidias con LML, tendremos que configurar el parser que usará nuestra aplicación. Para ello usaremos una sentencia similar a la que vemos a continuación:

Parser.java

```
1 | @Override
2 | protected LmlParser createParser() {
3 |     return VisLml.parser().actions("actions", Actions.class).i18nBundl
4 | }
```

Definición De Vistas

Para definir una vista (clase de Java) y enlazarlo con su correspondiente archivo `.lml` , tendremos

que declarar una clase que extienda de la clase abstracta `AbstractLmlView`. Dentro de esta clase definiremos un método con la siguiente signatura, en la que especificaremos la ruta del archivo `.lml`:

ViewDef.java

```
@Override
public FileHandle getTemplateFile() {
    return Gdx.files.internal("views/main.lml");
}
```

Inyección De Componentes En Código

LML define una serie de anotaciones para inyectar componentes definidos en los archivos `.lml` en el código Java y poder manipularlos ahí. El enlace se crea usando el valor del atributo `id` definido en el componente en el archivo `.lml`.

Injection.java

```
@LmlActor("tree")
private VisTree tree;
@LmlActor("path")
private VisTextField path;
@LmlActor("board")
private GridGroup board;
```

Disparadores De Acciones

LML define una serie de anotaciones que nos permiten indicar en los archivos `.lml` posibles acciones que disparen los elementos al ser activados una serie de eventos (botones, selección de texto, cambios en el componente…).

Las acciones pueden ser locales de la vista, en cuyo caso se especifica el nombre de la función que queremos disparar:

LocalAction.lml

```
<textbutton onclick="roll">Roll Dice</textbutton>
```

LocalAction.java

```
public void roll() {
    result.setText(String.valueOf((int) (MathUtils.random() * 1000)));
```

```
3 | }
```

O también se nos permite definir acciones globales. Para ello, definiremos una clase que implemente la interfaz `ActionContainer` y añadiremos anotaciones a cada función que definamos en esta clase:

GlobalAction.lml

```
1 | <textbutton onclick="setLocale" id="{element}" pad="10">
2 |     <:if test="{element} == en">
3 |         English
4 |     </:if>
5 |     <:if test="{element} == es">
6 |         Espa&ntilde;ol
7 |     </:if>
8 | </textbutton>
```

GlobalAction.java

```
1  | public class Actions implements ActionContainer {
2  |     private final GdxUI app = (GdxUI) Gdx.app.getApplicationListener()
3  |
4  |     @LmlAction("setLocale")
5  |     public void setLocale(final Actor actor) {
6  |         final String localeId = LmlUtilities.getActorId(actor);
7  |         final I18NBundle currentBundle = app.getParser().getData().get
8  |
9  |         if (currentBundle.getLocale().getLanguage().equalsIgnoreCase(l
10 |             return;
11 |
12 |         app.getParser().getData().setDefaultI18nBundle(I18NBundle.crea
13 |         app.reloadViews();
14 |     }
15 | }
```

Soporte Para Internacionalización

LML tiene soporte para internacionalización (bundles de i18n).

Para configurarlo, tendremos que especificar en la configuración del parser de LML el bundle de i18n

que queremos usar:

```
i18n.java
1 return VisLml.parser().i18nBundle(I18NBundle.createBundle(Gdx.files.ir
```

Dicho bundle se debe situar en la carpeta `assets` , en un subdirectorio llamado `i18n` y debe contener archivos nombrados con el siguiente formato:
`<nombre_bundle>_<idioma>.properties` , ej: `bundle_en.properties` .

El contenido de los bundles ha de tener el siguiente formato:

```
bundle_en.properties
1 windowTitle=Compass
2 helloWorld=Hello World !
3 randomPrompt=Click to roll a number:
4 language=Language
5 changeLanguage=Change language !
6 back=Go back
7 exit=Exit
8 website=Visit website
```

Todos los diferentes lenguajes que pertenezcan a un mismo bundle deberán declarar exactemante las mismas `Strings` , para que no existan problemas a la hora de encontrarlas y usarlas en los archivos `.lml` .

Para usar una cadena de texto definida en un bundle en un archivo `.lml` , usaremos la siguiente notación:

```
BundleString.lml
1 <linklabel href="https://neko250.github.io">@website</linklabel>
```

LML.dtd

LML nos permite crear y almacenar un archivo `.dtd` (Data Type Definition), en el que se definen los elementos que podemos usar en los archivos `.lml` , así como los atributos correspondientes.

También define una serie de macros que pueden servir para facilitar la definición de los archivos `.lml` . Entre ellas encontramos `<:for element="a;b;c">` , para realizar una serie de iteraciones; o `<:if test="{element} == asdf">` para realizar comprobaciones lógicas, por ejemplo.

Conclusiones

El desarrollo de interfaces gráficas en LibGDX sería bastante obtuso si no fuera por la existencia de addons como LML o VisUI (similar a la construcción de interfaces con Swing). Gracias a este conjunto de herramientas, la construcción de interfaces se hace de manera modular, separando los diferentes dominios lógicos que abarca: interfaz, código de manipulación de la interfaz, código de backend, internacionalización…

Nuestra opinión personal sobre el desarrollo de interfaces gráficas con este set, concluye remarcando la facilidad, escalabilidad y potencia a la hora de trabajar con él.