Clickteam Fusion 2.5

Android Extension SDK

Fernando Vivolo - October 25, y

Android SDK

Introduction

Welcome to the Clickteam Fusion Android extension SDK. Please read this file up to the end. After reading it, you will be able to create new extensions for the Fusion2 Android runtime. If you are fluent with the extension kit in other languages (like Java or C), you will see that the Android extension kit is very similar.

This extension kit contains all the source code of the Android runtime. We would be really happy to hear any suggestion you have about this source code, as it might help us improving it. If you have suggestions or questions about the source code, please send a mail to :

fernando@clickteam.com (or yvlam@clickteam.com)

Of course, the source code of the Android runtime is NOT in public domain, or under any kind of GNU license. It remains the property of Clickteam, and you should not copy it or distribute it in any way.

Getting ready to program.

First of all please be sure you have installed on your computer the JDK, the Google Android SDK and Eclipse or Android Studio. You will also need Fusion 2.5 (build 285.1 or above) and its Android exporter (make sure you run the Fusion update patch after you install the Android exporter).

You will also need first to make the Windows version of your extension (MFX file). If you have no MFX file for your extension yet, please use the Fusion 2.5 SDK to make one. The MFX file is a DLL that contains the definition of your actions, conditions, expressions, properties for the editor (and for the Windows runtime if you want your extension to be available in Windows applications). The Fusion 2.5 SDK is available on clickteam.com, either in the Download section or in the private Product Owner's Lounge forum, Fusion 2.5 section. This Fusion Android SDK only explains how to build the Android version of your extension, i.e. the routines that will be executed at runtime on Android devices.

The first thing you need to do, is to copy the "RuntimeAndroid.zip" file on your computer and import it via Eclipse or Android Studio. If you use Eclipse open the project and let it refresh. You are ready to begin development.

Creating a new extension.

To create a new extension :
• Right click on the "Extensions" folder in the project window.
• Choose "New file"

• Choose "Class" and click on it
• Enter for the name of your extension, the name "CRun" followed by the file name of your extension (the name is case sensitive). For example if the extension name is Android then the name of the class should be "CRunAndroid"
• Start programming your extension.

You will have to "link" the Fusion Android runtime with your extension, so that it can be created by the runtime. To do so :

• Open the "Extensions / CExtLoad.java" file

• At the beginning of the file, between the //STARTCUT and //ENDCUT markers, insert this small code:

```
if ( name.compareToIgnoreCase("Android")==0 )
{
        object=new CRunAndroid();
}
```

Replace both occurences of Android by the filename of your extension. These lines will incorporate your extension in the runtime.

Your extension must reside in one .java file. You are free to create more than one classes in this source file or add external files if necessary.

Please note that the sources of the Fusion runtime included in this SDK contain all the basic objects, as well as the Easing and Android objects. If you need another object, please unzip its zip file (from the data/runtime/Android folder of Fusion) into the project folder and add the creation code in CExtLoad.java as you did for your extension. If you need to see example extensions, just have a look at all the files in the extensions folder.

Publishing your extension

When you have programmed your extension (see the Extension Class part below), it's time to add it to Fusion. To do so :
• Compress your extension, including any resources, layout, libs, etc. The pathnames must be relative to the project folder (e.g. your extension java file must be in src\extension in the zip file). Here is for example the content of the zip file of the List object (kclist.zip):

kclist.zip

Name	Type	Modified	Size	Packed	Path
☐ list_selector.xml	XML Document	10/5/2015 4:56 PM	674	223	res\drawable\
☐ fusion_simple_list.xml	XML Document	10/13/2015 9:39 PM	1,136	616	res\layout\
☐ fusion_list.xml	XML Document	10/13/2015 9:39 PM	119	96	res\values\
☐ CRunkclist.java	JAVA File	10/13/2015 10:09 ...	22,9...	5,700	src\Extensions\

- And then copy your zip file to the Data\Runtime\Android folder of Fusion. Fusion will now detect that your extension is compatible with the Android exporter and will include it when you build an Android APK file.

Debugging your extension

Create your test application in Fusion, and select "Android application" as build type. Then build your application in the RuntimeAndroid folder and press **Shift** while building your application: Fusion will replace the application.ccn file in the res\raw sub-directory of this folder, as well as all the sounds of your test application. And then compile and run your project in Eclipse or Android Studio to debug it.

The extension class

As stated above, the name of your class should be "CRunNameOfExtension", and it should be in the Extensions folder.

Derive your CRunNameOfExtension class from the **CRunExtension** class, or if it's based on Views from the **CRunViewExtension** class which is at the end a mid layer and is also derived from **CRunExtension**.

The class contains functions similar to the C++ extensions, for the runtime. Lets see the functions in details.

In the Android runtime, all the object names start with "C". The minimal #import line needed in the .java file for your extension to compile are the following:

```
import Actions;
import Application;
import Conditions;
import Expressions;
import OI;
import Params;
import RunLoop;
```

```
import Runtime;
import Services;
```

Default variables

Two variables are defined in the parent parent of the object :
• **ho** : points to the CExtension object. Equivalent to the headerObject structure in Fusion. **ho** is also useful as callback functions are defined here to exchange data with the main runtime.
• **rh** : points to the CRun object. Equivalent to the runHeader structure in Fusion.

Constructor

Nothing special to do in the constructor of the extension. But you can have your own code in it. Remember it's called during the onCreate time of the activity.

public int getNumberOfConditions()
This function should return the number of conditions contained in the object (equivalent to the CND_LAST define in the ext.h file).

public boolean createRunObject(CBinaryFile file, CCreateObjectInfo cob, int version)
 or (for a View)
public void createRunView(CBinaryFile file, CCreateObjectInfo cob, int version)
This function is called when the object is created. As Android objects are just created when needed, there is no EDITDATA structure as in a Windows extension. Instead, values are send via a CFile object, pointing directly to the EDITDATA of the object in the application file.
The CFile object allows you to read the data. It automatically converts PC-like ordering into Android ordering. It contains functions to read bytes, shorts, int, colors and strings. Read the documentation about this class at the end of the document. "Version" contains the version value contained in the extHeader structure of the EDITDATA. So all you have to do, is read the data from the CFile object, and initialize your object (accordingly to the version if required). Return true if your object has been created successfully.

public void destroyRunObject(boolean bFast)
Called when the object is destroyed. This routine should free all the memory allocated by the object. bFast is true if the object is destroyed at end of frame. It is false if the object is destroyed in the middle of the application, please take a look at the Edit example in case you need to destroy a view in the middle of the application.

public int handleRunObject()

Same as the C++ function. Do all the tasks needed for your object in this function. As the C function, this function returns a value indicating what to do :

- REFLAG_ONESHOT : handleRunObject will not be called anymore
- REFLAG_DISPLAY : displayRunObject is called at next refresh.
- Return 0 and the handleRunObject method will be called at the next loop.

public void displayRunObject()

Called to display the object. If your object needs to be displayed on the screen, you need to add it to one of the display lists. Good examples are ActiveSystemBox and BackgroundSystemBox, their java sources are KcBoxA and KcBoxB.

public void pauseRunObject()

Called when the application goes into pause mode. In Android this is associated with the OnPause method from LifeCycle

public void continueRunObject()

Called when the application restarts from background, associated with OnResume.

public CFontInfo getRunObjectFont()

Equivalent to the C++ version. This function returns a CFontInfo object, a small object equivalent to the LOGFONT structure in C. See at the end of the document the definition of the CFontInfo object.

public void setRunObjectFont(CFontInfo fi, CRect rc)

Called when the font of the object needs to be changed. The rc parameter is null when no resize is needed.

public int getRunObjectTextColor()

Returns the current color of the text as an Integer.

public void setRunObjectTextColor(int rgb)

Sets the current color of the text.

public CMask getRunObjectCollisionMask(int flags)

If implemented, this function should return a CMask object that contains the collision mask of the object. Return null in any other case.

public boolean condition(int num, CCndExtension cnd)

The main entry for the evaluation of the conditions.

• num : number of the condition (equivalent to the CND_ definitions in ext.h)

• cnd : a pointer to a CCndExtension object that contains useful callback functions to get the parameters. This function should return true or false, depending on the condition.

public void action(int num, CActExtension act)

The main entry for the actions.

• num : number of the action, as defined in ext.h

• act : pointer to a CActExtension object that contains callback functions to get the parameters.

public CValue expression(int num)

The main entry for expressions.

• num : number of the expression

To get the expression parameters, you have to call the getExpParam method defined in the "ho" variable, for each parameter. This function returns a CValue* which contains the parameter. You then do a getInt(), getDouble() or getString() with the CValue object to grab the actual value.

This function returns a pointer to a CValue object containing the result. The content of the CValue can be a integer, a double or a String. There is no need to set the HOF_STRING

flags if you return a string : the CValue object contains the type of the returned value.

You should alloc the CValue yourself, a good place is the constructor and set the initial value to Cvalue(0).

Callback functions

The "ho" variable in the extension object gives you access to the CExtension object, which is derived from the main CObject class. A few callback functions are available in the CExtension object. Here are these functions :

public int getX()

Returns the current X coordinate of the object (hoX).

public int getY()

Returns the current Y coordinate of the object (hoY)

public int getWidth()

Returns the current width of the object (hoImgWidth).

public int getHeight()

Returns the current height of the object (hoImgHeight).

public void setX(int x)

Changes the position of the object (hoX), and takes care of the movement and refresh. Same remark as setPosition.

public void setY(int y)
> Same as setX for Y coordinate. Same remark as setPosition.

public void setWidth(int width)
> Change the width of the object, taking care of the hoRect fields.

public void setHeight(int height)
> Same as setWidth, for height.

public void setPosition(int x, int y)
> Change the X, Y position of the object

Load Images from Editor

public void loadImageList(short[] handles)
> This method should be called in the createRunObject method of your object. If your object uses images stored in the image bank, you must call this method so that the proper images are loaded.
>
> Just make an array with all the handles of the images, the size should be the exact number of images to load. Call this method (it may take some time to return). All the images will be loaded in the runtime.
>
> Here a small example:

```
ImagesNumbers = file.readShort();
IconImages = new short[ImagesNumbers];

if(ImagesNumbers > 0) {
        int i;
        for (i = 0; i < ImagesNumbers; i++)
        {
                IconImages[i] = file.readShort();
        }

        ho.loadImageList(IconImages);
}
```

public CImage getImageFromHandle (short handle)
> Call this function to retrieve an image from a handle. The image must have been previously loaded with loadImageList. The value returned could be null if the image could not be loaded, but this is very unlikely to occur. CImage is part of the "Banks" package. So you should import it with "import Banks;".

public void reHandle()

If you returned a REFLAG_ONESHOT value in your handleRunObject method, this will force the method to be called at the next loop.

public void generateEvent(int code, int param)
Generate an event with the specific code. The parameter can be retrieved with the ho.getEventParam() method. You should prefer the pushEvent method, specially if your event is generated in another thread than the main thread.

public void pushEvent(int code, int param)
This is the method of choice to generate an event.

public void pause()
Pauses the application, when you have some lengthy work to perform (like opening a file selector).

public void resume()
Resumes the application at the end of pause.

public void redraw()
Forces a redraw of the object (the displayRunObject routine is called at next refresh).

public void destroy()
Destroys the object at the end of the current loop.

public int getExtUserData()
Returns the private field of the extHeader structure.

public void setExtUserData(int data)
Changes the private field of the extHeader structure.

public int getEventCount()
Returns the rh4EventCount value, used in controls to trigger the events.

public CValue getExpParam()
Returns the next expression parameter.

public CObject getFirstObject()
Returns the first object currently defined in the frame. Should be used in conjunction with getNextObject().

public CObject getNextObject()
  Returns the next object in the list of objects of the frame. Returns null if no more objects are available. This method will return the extension object it is called from.

CRun callback functions

  You access the CRun object via the "rh" variable defined in the CRunExtension class. This object contains three methods used to define global data.
  Some extensions need to communicate between objects. In C++ this can be done simply by defining a global variable in the code. In Android there are three functions in CRun to allow you to create global classes.

public void addStorage(CExtStorage data, int id)
  Adds a new object to the storage, with the "id" identifier. "id" is a simple integer number. The storage class must be derived from the class CExtStorage. This function has no effect if an object with the same identifier already exists.

public CExtStorage getStorage(int id)
  Returns the storage object with the given identifier. Returns null if the object is not found.

public void delStorage(int id)
  Deletes the object with the given identifier.

CActExtension callback functions

  The CActExtension object is transmitted to the extension "action" method when a action is called. This object contains callback functions to gather the parameters of the action. These functions want two parameters:

   • The CRun object (available in the extension as "rh").
   • The number of the parameter in the action, starting at 0

public CObject getParamObject(CRun rhPtr, int _num)
  Returns the CObject pointed to by the PARAM_OBJECT.

public int getParamTime(CRun rhPtr, int _num)

Returns the time value in milliseconds.

public short getParamBorder(CRun rhPtr, int _num)
Returns the border parameter.

public short getParamDirection(CRun rhPtr, int _num)
(obsolete, but might be used in old extensions). Returns a direction from 0 to 31.

public int getParamAnimation(CRun rhPtr, int _num)
Returns the number of the animation.

public short getParamPlayer(CRun rhPtr, int _num)
Returns the number of the player.

public PARAM_EVERY getParamEvery(CRun rhPtr, int _num)
Returns a pointer to the eventParam structure containing the Every parameter. Look for the
definition in Cevents.Java. All param structures are located in Params.

public int getParamSpeed(CRun rhPtr, int _num)
Returns a speed, from 0 to 100.

public CPositionInfo getParamPosition(CRun rhPtr, int _num)
Returns the X and Y position contained in the parameter. Use the class CPosition to get the X
and Y positions.

public short getParamJoyDirection(CRun rhPtr, int _num)
Returns a joystick direction.

public PARAM_ZONE getParamZone(CRun rhPtr, int _num)
Returns a pointer to the PARAM_ZONE parameter. The parameter is defined like this:
short X1;
short Y1;
short X2;
short Y2;

public int getParamExpression(CRun rhPtr, int _num)
Returns the value contained in the expression.

public int getParamColour(CRun rhPtr, int _num)
Returns a color, as an integer.

public short getParamFrame(CRun rhPtr, int _num)
> Returns a number of frame.

public int getParamNewDirection(CRun rhPtr, int _num)
> Returns a direction, from 0 to 31.

public short getParamClick(CRun rhPtr, int _num)
> Returns the click parameter (left/middle/right button).

public String getParamExpString(CRun rhPtr, int _num)
> Returns the result of a string expression. You should copy the string as it.

public double getParamExpDouble(CRun rhPtr, int _num)
> Returns as a double the result of the evaluation of the parameter.

CCndExtension callback functions

The CCndExtension object is transmitted to the extension when calling the condition method. It contains callbacks to gather the parameters of the condition. Most of the methods are identical to the ones in CActExtension, with the following differences :
> Missing methods: getParamPosition
> Different returns
> The getParamObject method returns a pointer to the EventParam structure (as in the C++).

public boolean compareValues(CRun rhPtr, int _num, CValue value)
> In the C++ version, when you had a PARAM_COMPARAISON parameter in the condition, the condition routine returned a long value, and Fusion was automatically doing the comparison with the parameter. You have to call this method in the Android version. For example, for a condition with a PARAM_COMPARAISON as first parameter, the end of the condition method should be:
> return compareValues(rhPtr, _num, value);
> Where value is a CValue object containing the value to compare with.

public boolean compareTime(CRun rhPtr, int _num, int t)
> Same as the previous function, for PARAM_CMPTIME parameters, where "t" is the time to compare to.

Useful objects

Below are described some of the objects of the runtime you will be using to program your extension.

The CFile object

This object is passed in the createRunObject method. It automatically points to the start of the extension data (EDITDATA structure). It automatically performs the indian translation between PC values and Android values.

public native int skipBytes(int n);
Skips bytes in the file.
public native int getFilePointer ();
Returns the current file pointer position.
public native void seek (int pos);
Changes the file pointer position.
public native byte readByte();
Reads one byte.
public char readAChar()
Reads a char (2 bytes).
public void readAChar(char[] b)
Reads an array of char.
public native short readAShort();
Reads a short integer (two bytes).
public int readAInt()
Reads an integer (4 bytes).
public int readAColor()
Reads a color, making it compatible with Android color values (inversion of Red and Blue values).
public String readAString()
Reads a string that finishes with 0. The string is allocated for you, you should release it later in the code.
public String readAString(int size)
Reads a string of the given size. The string is allocated for you, you should release it later in the code.
public CFontInfo readLogFont()
Reads a LOGFONT structure into a CFontInfo object. The object is allocated for you, you should release it later in the code.

public CFontInfo readLogFont16()
Old Fusion objects only. Reads an old 16-bit LOGFONT structure into a CFontInfo object. The object is allocated for you, you should release it later in the code.

The CValue object

The CValue object is used in expression evaluation. It can contain an Integer, a double or a String.

public CValue(int i)
Creates the object as integer with the given value.

public CValue(double d)
Creates the object as double with the given value.

public CValue(String s)
Creates the object as string with the given value.

public byte getType()
Returns the type of the object. The return value can be :
TYPE_INT : an integer
TYPE_DOUBLE : a double
TYPE_STRING : a string

public int getInt()
Returns an integer. (if the object is of TYPE_DOUBLE, converts the value to int).

public double getDouble()
Returns a double.

public String getString()
Returns the string. Warning, if the object is of type int or double, this will NOT convert the value to string.

public void forceInt(int value)
Changes the type of the object to int and sets its value.

public void forceDouble(double value)
Changes the type of the object to double and sets its value.

public void forceString(String value)
Changes the type of the object to string and sets its value.

public void forceValue(CValue value)
Forces the content and type of the CValue.

public void setValue(CValue value)
Changes the content of the object, doesn't change its type.

The CFontInfo object

The CFontInfo object is a replacement of the LOGFONT structure in C++. It contains the name of the font, its height, weight and attributes.
Fields contained in the object :
 int lfHeight;
 int lfWeight;
 unsigned char lfItalic;

unsigned char lfUnderline;
unsigned char lfStrikeOut;
String lfFaceName;

The CRect structure

This structure is intended as a replacement of the C++ RECT structure. Fields :
int left;
int top;
int right;
int bottom;

Java functions defined for CRect :

public void load(CFile file)
Loads a CRect structure from a Cfile.
public void inflateRect(int dx, int dy)
Grows the size of the given CRect.
public CRect()
Returns an empty CRect.
public boolean ptInRect(int x, int y)
Returns true if the point lies within the CRect surface.
public boolean intersectRect(CRect rc)
Returns true if the two CRect intersetcs.

The CPoint structure

The CPoint structure is a replacement of the C++ POINT structure. Is only contains two fields:
int x;
int y;

The Renderer class

The render class depends on the opengl version the user chose, here are some common features:

public native void clear (int color);
Clears the current render target (either the screen or the currently bound texture)
protected void scissor (int x, int y, int w, int h)
Sets the Scissor rectangle to begin at 'x', 'y' with the given 'width' and 'height'.
No drawing can be performed outside the currently scissor rectangle until it is reset
again with a call to 'scissor(false)'.

protected void scissor (boolean flag)
> Resets the current scissor rectangle and drawing is allowed over the entire screen or RenderToTexture.

Drawing

public abstract void renderImage(ITexture image, int x, int y, int w, int h, int inkEffect, int inkEffectParam);
> Renders the previously set image (CImage) to the given x,y coordinate with the width and height and inkeffect.

The most common image drawing routine that takes an image, x, y, width,height and inkeffect parameters and draws the image there immediately.

This routine does not take hot-spots into account so the hotspot must be subtracted from the position before passing the position to this routine.
> Ink effect values:
> The inkEffect variable contains both the ink-effect to use and some flags.
> You can set the ink-effect to these values:
> BOP_BOPY
> BOP_BLEND
> BOP_INVERT
> BOP_ADD
> BOP_SUB
> If the BOP_RGBAFILTER flag is not set in the inkEffect variable then the effectParam value will be treated as a blending value between 0 and 128.
> If the BOP_RGBAFILTER flag is set, then the effectParam value should contain an ARGB color. (Alpha first, then RGB value f
> Example 1: Draw an image using 50% semitransparency
>> public abstract void renderImage(image, 0, 0, 32, 32, BOP_BLEND, 64);
> Example 2: Draw an image with a red blending color and 50% semitransparency
>> public abstract void renderImage(image, 0, 0, 32, 32, BOP_BLEND|BOP_RGBAFILTER,

Color.argb(128,255,0,0));

public abstract void renderScaledRotatedImage(ITexture image, float angle, float sX, float sY, int hX, int hY, int x, int y, int w, int h, int inkEffect, int inkEffectParam);
> Works the same way as 'renderImage' with a few differences: it takes an angle and a hotSpot position around which the image will be rotated.

public abstract void renderPattern(ITexture image, int x, int y, int w, int h, int inkEffect, int inkEffectParam);
> Draws a pattern of the given 'image' starting at (x,y) and will be repeated continously inside the width and height.

The (x,y) denotes the top-left corner of the rectangle inside which it will draw.

Note: See 'renderImage' for an explanation of how to use the ink-effects.

public abstract void renderPatternEllipse(ITexture image, int x, int y, int w, int h, int inkEffect, int inkEffectParam);

Same as 'renderPattern' but is masked out by the shape of an ellipse that fits inside the width and height.

The (x,y) denotes the top-left corner of the rectangle inside which it will draw.

See 'renderImage' for an explanation of how to use the ink-effects.

public abstract void renderPoint(ITexture image, int x, int y, int inkEffect, int inkEffectParam);

Identical to 'renderImage' except it only draws a single 1x1 pixel. Gives a speedup when many of these are drawn.

public abstract void renderLine(int xA, int yA, int xB, int yB, int color, int thickness);

Draws a line from (xA,yA) to (xB,yB) with the given color and thickness.

public abstract void renderGradient(int x, int y, int w, int h, int color1, int color2, boolean vertical, int inkEffect, int inkEffectParam);

Draws a rectangular gradient from (x,y) to (x+w, y+h) using the given ink-effect.

The 'colors' array should contain 4 ARGB colors (16 unsigned chars) each color being one of the corner colors.

Note: The order of the colors are as follows:

top-left, top-right, bottom-left, bottom-right (follows a Z pattern).

Note: See 'renderImage' for an explanation of how to use the ink-effects.

public abstract void renderGradientEllipse(int x, int y, int w, int h, int color1, int color2, boolean vertical, int inkEffect, int inkEffectParam);

Draws an ellipse shaped gradient contained within the rectangle that goes from (x,y) to (x+w, y+h) using the given ink-effect.

The 'colors' array should contain 4 RGBA colors (16 unsigned chars) each color being one of the corner colors in the bounding rectangle.

Note: The order of the colors are as follows:

top-left, top-right, bottom-left, bottom-right (follows a Z pattern).

Note: See 'renderImage' for an explanation of how to use the ink-effects.

public abstract void renderRect(int x, int y, int w, int h, int color, int thickness);

Draws a simple rectangle from (x,y) to (x+w, y+h) with a fill color and line thickness.

The CTextSurface class

A buffer class to prevent text from being drawn constantly by buffering it inside a CImage class. Use this class whenever you want to draw text to the screen.

There are two modes of drawing text with this class: 'Single string', and 'manual'.

Single string:

If you only want to draw a single string somewhere in your extension you can use this method as it is the simplest. In your extension's drawing routine you simply call the 'setText' routine of the

CTextSurface with the text, flags, color and font. CTextSurface will itself check if it is necessary to redraw the text in the buffer and upload it to the graphics card automatically. After the string is set you can simply call 'draw' with the given position and ink-effect to draw it wherever you like.

Manual:

If your extension draws multiple strings at one time (for example the Date & Time object in analog clock mode draws 12 strings in a circular fashion) the manual mode will be the right routine to use. It involves a little more work:

- You first have to manually clear the text surface using 'manualClear'
- You then call 'manualDrawText' as many times as you like to draw text wherever you want.
- You must finally call 'manualUploadTexture' before you call the 'draw' method.

Note: This mode does not automatically detect whenever it is necessary to update and renew the text to the graphics card so you must try to keep this to a minimum. For an example the Date & Time object only updates the text whenever it is necessary (typically only once at the creation time of the object).

Objects using the CTextSurface (for reference):
• String object
• Score object
• Counter object
• Active and Background System Box
• Time and Date object (for the visual analog and digital clock)
• Hi-Score object

Derived Activity Method

In CRunExtension you'll find some derived methods executed during the lifecycle of an activity. To see an example extract the source code of the Admob object.