Clickteam Fusion 2.5 HTML5 extension SDK

By François 20/01/2014

Welcome to the Fusion 2.5 HTML5 extension SDK. Please read this file up to the end. After reading it, you will be able to create new extensions for the Clickteam Fusion 2.5 HTML5 runtime. If you are fluent with the extension kit, you will see that the HTML5 extension kit is very similar to the Flash, XNA and iOS extension kit.
This 2014 version replaces the last version published in 2012(!). It is much easier to use.

Getting ready to program.

A source editor

You will need a source editor to write and edit the javascript file of your extension. I personally use Visual Studio 2012 (there is a free version, available here : http://www.microsoft.com/en-us/download/details.aspx?id=34673). It is a great and stable editor, that recognizes Javascript syntax.

The developpement cycle

1. Making the Windows version of the object

CT Fusion is a Windows-based application. All the extension objects displayed under the various editors are .MFX files, contained in the "Extensions" directory of your installation of Fusion. MFX files are Windows DLLs, that can be produced by a C++ compiler.
Before you start working on an HTML5 version of your extension, you must have created the Windows version of it. It will contain the definition of all the properties, conditions, actions and expressions.
Please refer to the Fusion 2.5 Windows Extension SDK for more information.

2, Creating your HTML5 extension

This extension kit contains the Javascript source code of an empty extension template. All the functions are implemented and commented.

- Copy the file Template.js file to your own extension development directory
- The name of the js file must be renamed to match the name of your extension. Let's imagine that your extension is called "MyFabulousExtension" (and therefore, the .MFX file is called MyFabulousExtension.mfx), you have to rename the Javascript file to :

  MyFabulousExtension.js

  Warning : the name of the file is **case-sensitive**, it must be exactly the same as the name of the Windows extension, or you will get an error when running the HTML5 application in the browser.
- Open your source-code editor, and load the new extension
- Do a search / replace, replacing "CRunTemplate" by "CRunMyFabulousExtension" in the whole source code.
- Begin the implementation of your code.

3, Development process

The "Utils" directory of this SDK contains a small utility called "HTML5 Extension Helper.exe". This application has been specially designed to simplify your task as a developper.

When building an HTML5 application, Fusion 2.5 copies the Javascript files from the data\runtime\html5 directory located in the installation directory of your copy of Fusion 2.5 The source-code of your new extension must be present in this directory when you build a mfa from Fusion 2.5.

Without the the utility

- Prior to build 282 of Fusion 2,5, you will have to load the file "Extensions.js" from the html5 directory on your installation drive, and insert the following code :

```
if (this.name == "MyFabulousExtension")
	return new CRunMyFabulousExtension();
```

  The code must be inserted in the "loadRunObject" function of the "CExtLoad" object. Tip : just search for "ENDCUT" in the source, and insert the two lines above. These two lines ensure that the object is actually created by Fusion 2.5 Runtime in development mode.
  Starting at build 282, Fusion 2.5 automatically patches the name of all the new extensions in Extension.js : this step is un-necessary.

- **Loop**: work on the source code of your extension.
- Manually copy the extension.js source code in the data\runtime\html5 folder of your Fusion 2.5 installation
- Under Fusion, do a Build & Run of the mfa you are using for testing. Make sure the selected build type is "HTML5 Development" : Fusion will copy all the Javascript files from the installation directory into the "src" directory of the builld target, save the graphics and sounds in the "resources" folder, and then launch the default browser on the new html file.
- You are free to open the browser's debugger and work on your extension : HTML5 Development build mode leaves the sources untouched.
- Once debugging is complete, goto **Loop**.

With the utility

- At start of session, launch the utility.
  Upon first use, you will have to enter the location of your extension's Javascript source code and verify that the installation directory of your copy of Clickteam Fusion 2.5 is correct. The utility will automatically check the version of the software, and eventually, if the build number is below 282, poke the missing lines in Extensions.js, as you would have done manually
- Click on the "Start" button of the utility, and forget about it

- **Loop**: Work on your source code. When you save it, the utility detects a change in the file, and copies it immediately into the data\runtime\html5 folder of your installation of Fusion 2.5. The icon of the utility in the task-bar will flash shortly during the process.
- Go back to Fusion 2.5, Build & Run your application in HTML5 Development mode and work under the browser and debugger.
- Goto **Loop**.

As you can see, the small utility will remove the annoying task of copying the extension in a protected folder on the system drive each time you want to test it.

4, Things to know

- You will have access to all the sources of the HTML5 runtime when you build your mfa as HTML5 Development project. The sources will be located in the "src" directory of the build target.
- Starting at build 281.3 of the editor, only the necessary files are copied. Not all the files of the runtime. But still readable an editable.
- You are free to define any class or global function in your extensions source-code. Just make sure to choose wise class or function names to avoid any future clash with newer extensions (it is a good idea to preceed all the global functions, variable and classes by a given prefix, made out of your own name or the extension's name).
- All the classes used in your extension must lay in one single file.
- HTML5 Final projects are compiled using the Google Closure compiler (https://developers.google.com/closure/compiler/). The compiler is set to the highest level of compression and encryption. Please read the documentation of the compiler if you face compilation problems in HTML5 Final project mode.
- You will, when you start a session of work on your extension, perform a "Build & Run" from Fusion 2.5 to build the HTML5 application, launch the browser and debug your code.
  If you do not close the browser and the debugger after a debugging session, it is possible to keep the position of your breakpoints until the next test. When coming back to Fusion after modifying your Javascript code, do a simple "Build" of the mfa instead of a "Build & Run". Fusion 2.5 will simply replace the code of the HTML5 Application by your new code on the drive. Then go back to the browser and press F5 : your application will be reloaded with the new code, and will stop at the previous breakpoints.
  Please note that Firefox data cache has a tendancy to get stuck during this process : the modifications are not loaded, even if you press F5. The only solution here is to clear the cache. This problem never occurred to me with Chrome or Internet Explorer.
- You can perfectly test your application on multiple browsers at the same time. After the Build & Run, just copy the URL into the other browsers. The only thing you have to do is keep your mfa loaded in Fusion so that the local web-server keeps on running.
- Remember that the entire content of the data\runtime\html5 directory is open and readable. You have access to the source code of all the existing Javascript extensions. We suggest that you have a look at their code if you are stuck when programming your extension.
- ActionScript is very close to Javascript, it is quite easy to port the code from a Flash extension to Javascript. The only difference being in the display routines.A complete document on the conversion process is included in this SDK : "FlashToHTML5.pdf"

The various objects

The CRunExtension class

As stated before, the name of your extension class should be "CRunNameOfExtension".

The class is derived from the CRunExtension class, using the "CServices.extend" function. The class functions are similar to those of the other runtimes, like the Flash, Java and XNA runtimes.

Default variables

Two useful properties are defined in the parent class :

- ho : points to the CExtension object. Equivalent to the headerObject structure in MMF. ho is also useful as callback functions are defined to exchange data with the main runtime.
- rh : points to the CRun object. Equivalent to the runHeader structure in MMF.

Use the "this" operator to access these properties from your code : this.rh or this.ho

Constructor

The constructor of the object is called at the very beginning of the object creation.
If you convert your code from other runtimes, where the variables are automatically set to 0 or null (like in Flash), it is a good idea to manually set each property of your object to 0 in the constructor. This may avoid conversion problems.

***number* getNumberOfConditions()**

This function should return the number of conditions contained in the object (equivalent to the CND_LAST defined in the ext.h file of the Windows SDK).
Warning : if this value is different from the value returned by the Windows version of the extension, the HTML5 application will surely crash.

***boolean* createRunObject(file, cob, version)**

This function is called when the object is created.

file
The file parameter points to a CFile object that allows you to read the data of this object. The data consist of the content of the C++ EDITDATA structure saved to disc.
The CFile object automatically converts PC-like ordering (big or little Indian I cant remember) into HTML5 ordering. It contains functions to read bytes, shorts, int, colors and strings. Read the documentation about this class at the end of the document.
cob
Contains a CCreateObjectInfo object that includes further info about the object created.
version
Contains the "version" value defined in the extHeader structure of the EDITDATA.

***void* destroyRunObject(bFast)**

Called when the object is destroyed. Due to garbage collection, this routine should not have much to do, as all the data reserved in the object will be freed at the next GC. 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.

***number* handleRunObject()**

This function is called at each loop by the runtime. It should perform all the tasks needed for your object to work. It should return a value :
• CRunExtension.REFLAG_ONESHOT : handleRunObject will not be called anymore. You can, in this case, have the function called again by calling *this.reHandle();*
• 0 : handleRunObject function will be called at the next loop.

***void* displayRunObject(renderer, xDraw, yDraw)**

Called to display the object, if your object needs to be displayed on the screen.

Renderer
Contains the renderer object to call for drawing the object. This object is documented later in this file.
xDraw, yDraw
These values must be added to the coordinates you send to the renderer, so that the object is properly centered in his layer within the display.
Please note that if your object is not to be displayed (it does not contain the OEFLAG_SPRITE or OEFLAG_BACKDROP flags), the displayRunObject routine can still be called when the object is moved. In this case, the context variable will be null, and xDraw and yDraw equal to zero.

***void* pauseRunObject()**

Called when the application goes into pause mode.

***void* continueRunObject()**

Called when the application restarts.

***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.

***void* setRunObjectFont(font, rc)**

Called when the font of the object needs to be changed.

- **font** : a CfontInfo object containing infos about the new font to use
- **rc** : null if the object does not need to be resized, or a CRect object with the size of the object after the font is changed.

***number* getRunObjectTextColor()**

Returns the current color of the text as an Integer.

***void* setRunObjectTextColor(rgb)**

Sets the current color of the text.

***boolean* condition(num, cnd)**

The main entry for the evaluation of the conditions.
- **num** : number of the condition
- **cnd** : a CCndExtension object that contains useful callback functions to get the parameters.

This function should return true or false, depending on the condition.

***void* action(num, act)**

The main entry for the actions.
- **num** : number of the action
- **act** : pointer to a CActExtension object that contains callback functions to get the parameters.

***number_or_string* expression(num)**

The main entry for expressions.
- **num** : number of the expression

To get the expression parameters, you have to call the this.ho.getExpParam() method defined in the "ho" variable, for each one of the parameters. This function returns a value which contains the parameter (number or string). It is mandatory that you ask for each one of the parameters of the expression each time this function is called, even if one of the parameter is out of range.
This method should return the correct value or 0 (or "") if the parameters are incorrect. The returned value can be a number or a string. There is no need to set the HOF_STRING flags if your return a string as you would do in C++.

The CExtension object callback functions

The "ho" variable in the CRunExtension object gives you access to the **CExtension** object, which is a derivative of the main **CObject** class (defined in the "Objects.js" source-code). A few callback functions are avaiable in this object.
Just use "this.ho" to call these functions. Example : this.ho.getX()

getX()

Returns the current X co-ordinate of the object (hoX).

getY()

Returns the current Y coordinate of the object (hoY)

getWidth()

Returns the current width of the object (hoImgWidth).

getHeight()

Returns the current height of the object (hoImgHeight).

setPosition(x, y)

Changes the position of the object, taking the movement into account (much better than poking into hoX and hoY).

setX(x)

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

setY(y)

Same as setX for Y co-ordinate. Same remark as setPosition.

setWidth(width)

Change the width of the object, taking care of the hoRect fields.

setHeight(height)

Same as setWidth, for height.

loadImageList(imageList)

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 and call this function : all the images will be loaded in the runtime.

Example of use in CreateRunObject :

```
this.images = new Array[4];
var n;
for (n = 0; n < this.images.length; n++)
        this.images[n] = file.readAShort();
file.loadImageList(this.images);
```

getImage(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. It returns a CImage object.

```
var image = this.getImage[this.images[3]);
var w = image.width;
```

reHandle()

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

generateEvent(code, param)

Generate an event with the specific code.
code : the number of the condition as defined in your extension source
param : a parameter to transmit to the action
The parameter can be recuperated with the this.ho.getEventParam function. The function executes the event before returning.

pushEvent(code, param)

Generates an event at the end of the game loop. This function stores the information about the event, and returns immediately. The events are unpiled and called at the end of the game loop.

pause()

Pauses the application, when you have some lengthy work to perform (like opening an alert box), thus making sure the game's timer is halted while the application is stopped.

resume()

Resumes the application at the end of your work.

destroy()

Destroys the object at the end of the current loop.

getExtUserData()

Returns the private field of the extHeader structure.

setExtUserData(data)

Changes the private field of the extHeader structure.

getEventCount()

Returns the rh4EventCount value, used in controls to trigger the events.

getExpParam()

Returns the next expression parameter. The returned value can be a number or a string.

getFirstObject()

Returns the first object currently defined in the frame. The returned value is a pointer to an object derived from **CObject** (with for example properties like hoX and hoY).
It should be used in conjunction with getNextObject().

getNextObject()

Returns the next object in the list of objects of the frame, or null if no more objects are

available. This function will return the extension object it is called from.
This loop scans all the objects currently active of the frame :

```
var object;
for (object = this.getFirstObject(); object != null; object = this.getNextObject()
{
        … do what you want with the object
}
```

The CExtension class also contains properties that you can use:

pLayer
The layer object that contains your extension if it is displayed (a CLayer object)

bShown
A boolean value indicating if the object is hidden ot shown. Note that you do not need to test this value in your displayRunObject function : the function is not called if the object is hidden.

CRun callback functions

The **CRun** object is the main object of the runtime : it contains the the code of the main loop.
You access the **CRun** object via the "rh" variable defined in the **CRunExtension** class (just use "this.rh" as the extension class is a derivative from the CRunExtension class).
This object contains three methods used to define global data.
Some extensions need to communicate between objects. The simplest Javascript approach would be to create a global variable in your code. This method would work if only one Fusion 2.5 application that uses your object is playing on the HTML page.
You should use these functions so that your code still works when several Fusion 2.5 HTML5 applications are running on the same HTML page. A "storage" object added in one frame will be available in another frame.

addStorage(data, id)

Adds a new object to the storage, with the "id" identifier. "id" is a simple integer number. This function has no effect if an object with the same identifier already exists.

getStorage(id)

Returns the stored object with the given identifier. Returns null if the object is not found.

delStorage(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 function to gather the parameters of the action. These functions want two parameters:

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

The parameter objects are defined in the "Params.js" source code.

getParamExpression(rhPtr, num)

Evaluates and returns the value contained in the expression. Can return a string. This function should be called when the parameter of your action is PARAM_EXPRESSION or PARAM_EXPSTRING.

getParamObject(rhPtr, num)

Returns the CObject pointed to by the PARAM_OBJECT parameter.

getParamTime(rhPtr, num)

Returns the time value in milliseconds of a PARAM_TIME parameter.

getParamBorder(rhPtr, num)

Returns the border parameter of a PARAM_BORDER parameter.

getParamDirection(rhPtr, num)

(obsolete, but might be used in old extensions). Returns a direction from 0 to 31.

getParamCreate(rhPtr, num)

Returns a pointer to the PARAM_CREATE object (for future use maybe).

getParamAnimation(rhPtr, num)

Returns the number of the animation of a PARAM_ANIMATION parameter.

getParamPlayer(rhPtr, num)

Returns the number of the player (PARAM_PLAYER)

getParamEvery(rhPtr, num)

Returns the delay in milliseconds entered as a parameter in the every condition (PARAM_EVERY)

getParamKey(rhPtr, num)

Return the key code contained in the PARAM_KEY parameter (Javascript keycode are similar to Windows keycodes. Do a search on the Internet for "Windows virtual key codes" to find a list of the key codes).

getParamSpeed(rhPtr, num)

Returns a PARAM_SPEED speed, from 0 to 100.

getParamPosition(rhPtr, num)

Returns the evaluation of a PARAM_POSITION parameter, used in a "Set position" action. Returns a CPositionInfo object that contains :

- x : the X coordinate
- y : the Y coordinate
- layer : the number of the layer
- dir : the direction where to create / position the object

getParamJoyDirection(rhPtr, num)

Returns a joystick direction (PARAM_JOYDIRECTION).

getParamShoot(rhPtr, num)

Returns a pointer to the PARAM_SHOOT object contained in the action (for future use maybe).

getParamZone(rhPtr, num)

Returns a PARAM_ZONE object, that contains :

- x1
- y1
- x2
- y2

getParamColour(rhPtr, num)

Returns a color, as an integer (PARAM_COLOUR), in the form 0xRRGGBB.

getParamFrame(rhPtr, num)

Returns a number of frame (PARAM_FRAME)

getParamNewDirection(rhPtr, num)

Returns a direction, from 0 to 31 (PARAM_NEWDIRECTION).

getParamClick(rhPtr, num)

Returns the click parameter :

- value & 0xff : 0 = left click, 1 = right click, 2 = middle click
- value & 0x100 : double click

The next two functions are, in Javascript similar to the getParamExpression function (as there is no difference in the code between strings and numbers). I have left them in the Javascript code for compatibility reasons :

getParamExpString(rhPtr, num)

Returns the string value contained in an expression.

getParamExpDouble(rhPtr, 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 functions are identical to the ones in CActExtension, with the following differences :

Missing functions:
getParamPosition
getParamCreate
getParamShoot

Different returns
The getParamObject method returns a pointer to the PARAM_OBJECT object (as in the C++) and not a pointer to the CObject object.

compareValues(rhPtr, num, value)

In the C++ version, when you had a PARAM_COMPARAISON parameter in the condition, the condition routine returned a long value, and Fusion 2.5 was automatically doing the comparison with the parameter.
You have to manually call this method in the Javascript version. For example, a condition with a PARAM_COMPARAISON as first parameter, the end of the condition method should be:

return cnd.compareValues(rh, 0, returnValue);

Where returnValue is the value to compare with. compareValues returns a boolean (true or false).

compareTime(rhPtr, num, t)

Same as the previous function, for PARAM_CMPTIME parameters, where "t" is the time to compare to. compareTime returns a boolean (true or false).

Useful objects

I will document now some of the objects of the runtime you will be using to program your extension.

The CFile object

This object is sent to you in the createRunObject method. It automatically points to the start of the extension data (EDITDATA structure, in memory). It automatically performs the indian translation between PC values and Javascript values.

array **readBuffer(size)**

Reads a array of bytes and returns the Array.

readBytesAsArray(array)

Reads a array of bytes using the given array length.

void **skipBytes(n)**

Skips n bytes in the file.

number **getFilePointer()**

Returns the current file pointer position.

void **seek(pos)**

Change the file pointer position.

number **readAByte()**

Reads one unsigned byte.

number **readAShort()**

Reads an unsigned short (two bytes).

number **readAInt()**

Reads a signed integer (4 bytes).

number **readShort()**

Reads a signed short (2 bytes).

number **readAColor()**

Reads a color making it compatible with PC color values (inversion of Red and Blue values). The color is returned in the form of : 0xRRGGBB. Use this function when you want to read "COLORREF" entries of the EDITDATA structure.

string **readAString([size])**

Reads a string. If you provide the size parameter, then reads exactly the given size (the string may stop before if a zero is found, but the file pointer will always point to the end of

the buffer).

***string* readAStringEOL()**

Reads a string until the next CR/LF characters. The file is positionned immediately after the CR/LF. Works with both PC and Mac end of lines.

***CFontInfo* readLogFont()**

Reads a LOGFONT structure into a CFontInfo object. The object is allocated for you.

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.

Properties contained in the object :

lfHeight (number)
The height of the font. Please note that the actual size of the font on the screen may vary from the C++ version to the Javascript version by a couple of pixels.

lfWeight (number)
The weight of the font. 400 means a normal font. Over 400 a bold font. Below 400 a light font.

lfItalic (number)
Equal to 1 if the font is italic, 0 if not.

lfFaceName (string)
A string containing the name of the font.

Functions of the object :

***void* copy(source)**
Copies the content of the given CFontInfo object.

***string* getFont()**
Returns the browser-compliant string equivalent of this font, ready to be used for drawing. Example of string returned by this function : "italic 400 12px Arial"

The CRect object

This object is intended as a replacement of the C++ RECT structure.

Fields :

- left (number)
- top (number)

- right (number)
- bottom (number)

public void load(CFile file)

Reads a CRect from the CFile object.

copyRect(srce)

Copies the content of the given CRect object in the object.

ptInRect(x, y)

Returns a boolean. true if the given point is located in the rectangle.

intersectRect(rc)

Returns a boolean. true if the two rectangle intersect.

The CPoint object

The CPoint object is a replacement of the C++ POINT structure. Is only contains two fields:

- x (number)
- y (number)

The CImage object

The CImage object contains one image of the game. This image is loaded at the beginning of the frame if it was not present in memory at that time.
In your extension, a call to "loadImageList" in the createRunObject function will enforce the loading of a list of images.
Each image is refered to by a handle. You can get a CImage object from a handle in your extension by calling : this.ho.getImage(handle);
The images used in the game can be grouped into mosaics of images. Despite of this, there will always be one CImage object per individual image.

Properties :

- width : the width of the image
- height : the height of the image
- xSpot : the horizontal position of the hot-spot
- ySpot : the vertical position of the hot-spot
- xAP : the horizontal position of the action point
- yAP : the vertical position of the action point

If the application was built without putting the images into mosaics :

- img : contains an HTML Image object with the image

If the application was built with images in mosaïcs :

- mosaic : contains the index of the mosaic compound image in the CFrame object, a HTML Image object. You can get this object with :
  var mosaic = cImage.app.frame.mosaicHandles[cImage.mosaic];
- mosaicX : the horizontal coordinate of the image in the mosaic
- mosaicY : the vertical coordinate of the image in the mosaic

The Renderer object

All the graphic outputs of the Javascript runtime are done via the Renderer object. In the current version, the renderer object is the "**StandardRenderer**" object, that uses the 2D Canvas function to draw the graphics. The displayRunObject function of your extension is called with the Renderer object as a parameter.
We have in plan to add a WebGL renderer in the future. When we do this, we will transmit the WebGL Renderer object to displayRunObject. If your extension calls the renderer functions to draw itself, it will be directly compatible with the changes.
You can, if you want, get the "context" of the Canvas in the renderer object and draw directly in it. If you do that, your object may not be compatible with future versions of the HTML5 runtime.
Please report to the end of this chapter for info on the inkEffect and inkEffectParam parameters.

Properties

_context : the Canvas drawing context.

Functions

renderLine(x1, y1, x2, y2, color, thickness, inkEffect, inkEffectParam)

Draws a line between two points.

renderRect(x, y, width, height, color, thickness, inkEffect, inkEffectParam)

Draws an empty rectangle of the given thickness and color.

renderEllipse(x, y, width, height, color, thickness, inkEffect, inkEffectParam)

Draws an empty ellipse of the given color and thickness.

renderSolidColor(x, y, width, height, color, inkEffect, inkEffectParam)

Draws a rectangle filled with one color.

renderSolidColorEllipse(x, y, width, height, color, inkEffect, inkEffectParam)

Draws an ellipse filled with one color.

renderGradient(x, y, width, height, color1, color2, vertical, inkEffect, inkEffectParam)

Draws a rectangle filled with a gradient of colors between color1 and color2. The "vertical" parameter is a boolean indicating if the gradient is horizontal or vertical.

renderGradientEllipse(x, y, width, height, color1, color2, vertical, inkEffect, inkEffectParam)

Draw an ellipse filled with a gradient between color1 and color2. The "vertical" parameter indicates if the gradient must be horizontal or vertical.

renderPattern(image, x, y, width, height, inkEffect, inkEffectParam)

Draws a rectangle filled by the repetition of a CImage image.

- image : a CImage object.

renderPatternEllipse(image, x, y, width, height, inkEffect, inkEffectParam)

Draws an ellipse filled by the repetition of the image of a CImage object.

- image : a CImage object

renderImage(image, x, y, angle, scaleX, scaleY, inkEffect, inkEffectParam)

Draws a CImage object.

- image : a CImage object containing the image to draw
- x, y : coordinates where to draw the hot-spot of the image
- angle : the rotation angle of the image around its hot-spot, in degrees
- scaleX, scale Y : the scale of the image in both axis. A value of 1.0 leaving the size of the image unchanged.

Please note that the renderImage function takes into account images stored within mosaïcs.

renderSimpleImage(image, x, y, width, height, inkEffect, inkEffectParam)

Draws a simple Javascript Image object.

clip(x, y, width, height)

Pushes the context and clips the graphic output.

unClip()

Restores the context to what it was before being clipped.

The inkEffect and inkEffectParam parameters

Every function of the renderer object handles an optional inkEffect and inkEffectParam parameter. These parameters define the current effect applied to the graphic output.

inkEffect :

- (inkEffect & CRSpr.BOP_SMOOTHING) != 0 : turns on "imageSmoothing" in the context
- (inkEffect & CRSpr.BOP_RGBAFILTER) != 0 : the globalAlpha property of the context is set from the formula : `(((inkEffectParam >>> 24) & 0xFF) / 255.0);`
- (inkEffect & CRSpr.BOP_BLEND) != 0 : the globalAlpha property of the context is set from the formula : `((128 - inkEffectParam) / 128.0)`
- (inkEffect & CRSpr.BOP_MASK) contains the actual effect to apply to graphic drawing :
    - CRSpr.BOP_ADD : the composite property of the context is set to "lighter"
    - CRSpr.BOP_XOR : the composite property of the context is set to "xor"
    - Any other value : the composite property of the context is set to "source-over"

The CTextSurface object

You can, if you want to draw text on the screen, draw it directly in the context using the normal Canvas functions. If you do that, your extension may not be compatible with a future WebGL version of the runtime.
The CTextSurface object can be used to draw text on the screen. It creates a hidden drawing canvas where the text is initially drawn. You then draw this canvas on the screen when you need it. The CTextSurface allows you to justify the text horizontally on the right, center or left, and vertically on the top, center or bottom. You can print several lines of text, separated by CR/LF : it is therefore much more powerful than the default fillText functions of the Canvas.
Note that the CTextSurface is compatible with the graphic fonts produced by the MobileFont object.

Constructor: CTextSurface (application, width, height)

- application : the CRunApp object of your game. You can refer to it with : this.rh.rhApp
- width : the width of the hidden drawing surface. If the text is larger than this value, it will be clipped
- height : the height of the hidden drawing surface. If the text is higher than this value, it will be clipped

measureText(text, font)

Returns the width of a string on the screen.

- text : the text to measure
- font : a CFontInfo object containing the definition of the font

setText(text, flags, rectangle, font, color)

Draws a paragraph of text in the hidden surface after erasing the background.

- text : a string containing the text to draw
- flags : a set of flags indicating how to draw the text
    - CServices.DT_LEFT : the text is justified on the left (default)
    - CSercices.DT_CENTER : the text is centered horizontally

    - CServices.DT_RIGHT : the text is justified on the right
    - CServices.DT_TOP : the text is aligned on the top of the rectangle
    - CServices.DT_VCENTER : the text is centered vertically in the rectangle
    - CServices.DT_BOTTOM : the text is aligned on the bottom of the rectangle
- rectangle : a CRect object containing the dimensions of the rectangle where to draw the text
- font : a CFontInfo object containing the definition of the font to use
- color : the color of the text to draw

Once the text is drawn in the hidden surface, you need to call the **draw** function in your displayRunObject function.

manualClear(color)

Erases the hidden drawing surface, with the given color.

manualDrawText(text, flags, rectangle, color, font, relief, color2)

Draws the text in the hidden surface with an eventual 3D effect. Does not erase the hidden surface prior to doing that.

- text, flags, rectangle, color, font : same as for the setText function
- relief : applies a 3D effect to the text. This parameter can take the following values:
    - 0 : no 3D effect
    - 1 : the text is displayed with color2 one pixel below the main text
    - 2 : the text is displayed with color2 one pixel on the right and bottom of the main text
- color2 : the color used to draw the shadow of the text

resize(width, height)

Changes the size of the hidden surface.

draw(renderer, x, y, inkEffect, inkEffectParam)

Draws the hidden surface on the screen.

- renderer : the renderer object in which to draw the hidden surface
- x, y : the coordinates of the top-left corner of the surface
- inkEffect, inkEffectParam : the effects used to draw the surface (see the Renderer object)

The CRunControl object

This is a handy class to simplify the handling of controls from an extension. We use it for the list, combo, edit box objects and video objects.
If your extension is a HTML element created by the browser, instead of deriving your extension from CRunExtension, derive it from CRunControl. New functions will be added to your extension object.

setElement(element)
Use this function immediately after creating your control. Pass the control as a parameter. This function automatically positions the control at the object's position and sets its size. The control is automatically added to the Canvas.

setX(X)
setY(Y)
setPosition(X, Y)
Changes the position of the control. Also stores the new position in hoX and hoY.

setWidth(width)
setHeight(height)
setSize(width, height)
Changes the dimension of the control and stores the new dimension in hoImgWidth and hoImgHeight.

DestroyRunObject()
Removes the control from the Canvas.