Introduction

The core of the sprite system is not much more than a single script, CSprite. The sprites form a hierarchical structure, not unlike an XML document with parent and child nodes. Because of Lingo's somewhat unfortunate naming of inheritance concepts (parent scripts, child objects, ancestors etc), talking about both hierarchical structures and inheritance as we will in this document can be confusing. I'll use the more common terminology:
parent script = class
child object = instance
ancestor = base class
script with an ancestor = derived class
object with an ancestor = instance of a derived class

When I talk about parents and children, I mean it in the XML sense. I'll use "offspring" to signify "children, or children's children, etc". "Ancestor" means "parent, or parent's parent etc"

The Rendering Pass

Normally, you have a single sprite, which corresponds to the stage – the root node, in XML terms. All other sprites are offspring to this sprite. Each frame, you tell the root sprite to render. It first renders itself, then it tells its children (ordered by locZ) to do the same. They do the same thing, so the render command spreads through the structure until all sprites have rendered.

Rendering normally consists of the following steps (unless you choose to override it with some other method): calculate the output rect or quad, then making a call that blits the pixels from a source image, to the target image.

Calculating sprite rects/quads

(These are not the actual property names, these names are used for readability)
(This is a simplified account of the procedure)
The most basic properties of a sprite are the sourceSize (#point) and sourceRegPoint (#point). I could've used a sourceRect to store the same information, but for a number of reasons I chose this instead.
Calc the sourceRect from sourceSize and sourceRegPoint. Pass this rect to a number of transforms (#scale, #loc etc) to get an outputRect. If the parent sprite is the root sprite, and the root sprite would be at (0,0), scale 1:1, non-rotated etc, this outputRect would be the rect to render to on stage. But the code is prepared for the parent to have any shape. To calculate the final output, the sprite loops through its four corners, asking its parent where that loc would be in the parent's space. If the parent isn't the render target, the parent asks its own parent the same question, with the transformed loc. This is repeated until the render target is reached.

There are a number of cases where optimizations can be done, by keeping track of parents properties – if they are plain rect sprites or quads, if they're scaled or not. I've implemented some, I think they're probably sufficient for most cases.

There are many other aspects of this. For example, as soon as quads are involved, either in a parent sprite or in the sprite itself (after rotation, shearing etc) it gets more complicated.

Blitting

Most often, the blitting is done using copyPixels, but you can also choose to use my jbCopyPixels Xtra which adds lots of new inks, or some other Xtra or method.

The render target image is normally the root sprite's image, which points at (the stage).image, but you can choose to set the render target to any other sprite (as long as it's an ancestor). This is common for optimization purposes. For example, a window made with 30 sprites, which gets dragged around a lot but doesn't change its contents often, should render into a separate image. That way only one blit is needed per frame while dragging the window, instead of 30.
You could also wish to send the render output to a 3D texture instead of the stage.

Dirty rects

TODO: When something within a sprite is changed that requires it to be rendered again, like loc, rect, ink, rotation etc, a flag is set. In the start of the rendering pass, the sprite notifies its render target sprite about the need to render the affected area. The render target sprite finds out which other of its children are in this "dirty" area, and adds them to the list of sprites to render.

2D GUI in 3D

In order to add the ability to render into 3D, I've taken a quick and dirty road which isn't strictly OOP – I plug in an extra class, C3DSprite, under CSprite in the structure.
C3DSprite uses CSprite as the base class, and therefore all classes that use normally CSprite as a base class must use a special command in the "ancestor" creating line;
ancestor = script(getCorrectSpriteScript(a_paOptions)).rawNew()
so that the correct base class is used.
C3DSprite overrides many of the CSprite handlers – render(), setImage(), rotate() etc. Each sprite has a plane in the 3D world. When you setImage(), it creates a texture and uses it on the plane. rotate() just rotates the model.
The 3D worlds hierarchical structure, with parent and child nodes, is not used. It would probably give a performance boost, but I'd have to be extremely careful so it would work exactly as the 2D version.
Mouse and keyboard interactions work just as usual.
There are many optimizations to be done with texture management. The script C3DManager (that sets up cameras etc for use with C3DSprite) already contain some code for this, but much more can be added.

Behaviors

Use the addBehavior and removeBehavior functions to manage behaviors. When adding a behavior, CSprite will check for the presence of event handlers (mouse, key, stepframe etc), and automatically register the instance for notification of such events. When removing a behavior, it will be unregistered for events, and also automatically killed (unless you specifically tell it not to).

Interactivity

Mouse and keyboard events are handled in a similar way. The root sprite gets the system message, then it spreads it through the offspring structure the same way as the render() command.

Mouse

Sprites are not mouse sensitive unless you call the activateMouseEvents(propertyList with settings) handler. (NOTE: if you add a behavior, and it has the hander mouseEvent(),activateMouseEvents() will be called automatically). The mouse handling for a sprite is done in a separate script, CMouseReactSprite. It checks when the mouse enter/leaves the sprite's area, whether the mouse is inside or not when buttons are pressed/released, and tells CSprite about it for further handling (sending to behaviors etc).

Here are the current options when creating the mouse sensitivity object:

#PassButton TRUE/FALSE, default FALSE: stop checking button on underlying sprites
#PassButtonUp TRUE/FALSE, default FALSE: stop checking buttonUp on underlying sprites
#PassRoll TRUE/FALSE, default FALSE: stop checking enter/leave on underlying sprites
#Draggable TRUE/FALSE, default FALSE: allow user to drag the sprite around.
#CheckInsideMode: #Pixel/#Rect, default #Pixel. Check only the rect/quad when deciding if the mouse is inside the sprite, or check the pixel under the sprite as well.
#AlphaThreshold: 0-255. For #Pixel inside checks – how opaque must pixel be.
#DoubleClickTime: milliseconds, default 200. When does two clicks count as a doubleclick

To receive a mouse event, you must define the handler mouseEvent. Example:

```
on mouseEvent me, a_oSender, a_sbEvent
  --a_oSender is the sprite itself. If you need the mouse handling object, use a_oSender.
m_oMouseReact
  case a_sbEvent of
    #Enter: put "Mouse entered sprite"
    #Leave: put "Mouse left sprite"
    #Down: put "Mouse button down on sprite"
    #UpOutside: put "Mouse was first down on sprite, then up outside sprite"
    #Up: put "Mouse button down outside of sprite, then up inside sprite"
    #Click: put "Mouse button down on sprite, then up inside sprite"
    #StillDown: put "Sent each frame while the mousebutton is down on sprite"
  end case
end
```

You can easily send fake mouse positions and button activity to the system, and stop the real information from getting through. It doesn't care where the information comes from. Useful when creating demos.

Keyboard

Not very sophisticated at the moment. Define keyDown and keyUp handlers, and they will be called properly. The keyCode is sent as the first argument to both handlers (yes, with proper handling of which key was released). TODO: also send modifier keys here, allow filters so only certain keys are being put through to the handler, allow aliases so you can register interest in e.g. #KeyPad0 instead of the keyCode (whatever it is).

Deriving from CSprite

CSprite is a complex script, but it's main objective is just to copy its image into another image. Its functionality must be extended in order to achieve many of the things you want to do in a game or multimedia production.
This can be done by adding behaviors, but sometimes the most logical and convenient way can be to write a new sprite class, derived from CSprite. Think of it as if you could easily create new member types in the normal Director; if you'd want to do that to handle a certain task, then deriving from CSprite is the way to go.

An example is the CText class. It is a fairly large class, derived from CSprite, that acts much like a Text Member on the stage would do – it reacts to keyboard and mouse events, keeps track of where the cursor is, the current selection and renders itself when necessary.
It could have been written as a behavior, but it wouldn't have made much sense. The sprite IS a text sprite, it doesn't just HAVE text functionality.
The derived sprite can create child sprites as well – CText does this for displaying the rects marking selections in the text.

Widgets

I've built a small test widget library, including CWindow, CButton, CScrollbar, CDragBar, CMenu, CUpDown, CFrame, CFolderStrucureView, CFolderContentsView. All these are derived from CSprite one way or another – some through another widget class, like CWindow which is derived from CFrame.
This will be an area where I hope the community can help out. Each widget class created out there has the potential of being useful to a lot of people – and if they're well implemented, they can be used directly without anyone having to look at the code, and also be the base class for other widgets.

There are some rules that apply to all CSprite derivates (you can take a copy of the script CTemplateSprite to get started a little quicker):

Always use
callAncestor(#Function, me, argument)
instead of
ancestor.function(argument)

You have to declare the property m_bDisposing. Initialization is not necessary.

The new() and kill() functions must look like this:

```
on new me, a_paOptions
  ancestor = script(getCorrectSpriteScript(a_paOptions)).rawNew()
  -- do your own stuff here
  callAncestor(#new, me, a_paOptions)
  return me
end

on kill me
  m_bDisposing = TRUE
  --do your own stuff here
  ancestor = callAncestor(#kill, me)
  return 0
end
```

If you want the Property Inspector, vCorse editor and animation objects to work properly with your new class, you must define the properties that are public in this way:

```
on getPublicProps me
  pa = callAncestor(#getPublicProps, me)
  pa.addProp(#SomeNewProperty, <a list like those in propertyDescriptionList>)
  return pa
end
```

The list above should probably contain info about how the property's control should look in the Property Inspector, so you can make a more customized inspector for your class. (You can also remove properties from the list, making them non-public and non-editable). Other handlers to include:

```
on getOneProp me, a_sbProp
  case a_sbProp of
    #Class: return "CThisClassName"
    #SomeNewProperty: return yourVariableOrWhatever
  end case
  return callAncestor(#getOneProp,me,a_sbProp)
end
on setOneProp me, a_sbProp, a_mvVal
```

```
  case a_sbProp of
    #SomeNewProperty: yourVariableOrWhatever = a_mvVal
    otherwise: callAncestor(#setOneProp,me,a_sbProp,a_mvVal)
  end case
end
```