First Steps in GNUstep GUI Programming (2): NSWindow, NSButton

Nicola Pero n.pero@mi.flashnet.it

July 2000 AD

In this tutorial you will learn how to create a window in your application. We will put then a button in the window. *Warning:* You need GNUstep core libraries version 0.6.6 or later to try out the examples.

1 Points, Sizes and Rectangles

The GNUstep base library defines some useful structs for dealing with two dimensional geometry: `NSPoint`, `NSSize` and `NSRect`. It is worth to quickly review them here, before beginning.

1.1 NSPoint

A `NSPoint` is defined as a struct with two members, the x and y coordinate of the point:

```
typedef struct _NSPoint
{
  float x;
  float y;
} NSPoint;
```

So, to access the `x` and `y` coordinates of a `NSPoint` called `myPoint`, you just do as in `myPoint.x` and `myPoint.y`. Please note that the coordinates of a `NSPoint` are floats; so, they might be negative and/or fractionary.

To create a point with given `x` and `y` coordinates, you can use the function (macro) `NSMakePoint ()`, as in the following example:

```
NSPoint testPoint;

testPoint = NSMakePoint (10, 20);
NSLog (@"x coordinate: %f", testPoint.x); // 10
NSLog (@"y coordinate: %f", testPoint.y); // 20
```

It might be worth quoting the (predefined) constant `NSZeroPoint`, which is a point with zero x coordinate and zero y coordinate.

1.2 NSSize

An `NSSize` is a struct describing the size of a rectangle, regardless of its position.

```
typedef struct _NSSize
{
  float width;
  float height;
} NSSize;
```

Using `NSSize` is completely analogous to using `NSPoint`, as in the following code example:

```
NSSize testSize;

testSize = NSMakeSize (0.5, 51);
NSLog (@"x coordinate: %f", testSize.width); // 0.5
NSLog (@"y coordinate: %f", testSize.height); // 51
```

`NSZeroSize` is a constant equal to a size with zero width and zero height.

1.3 NSRect

An `NSRect` is a struct describing both the position and the size of a rectangle:

```
typedef struct _NSRect
{
  NSPoint origin;
  NSSize size;
} NSRect;
```

Using `NSRect` is similar to using `NSPoint` and `NSSize`:

```
NSRect testRect;

testRect = NSMakeRect (8.1, -3, 10, 15);
NSLog (@"x origin: %f", testRect.origin.x); // 8.1
NSLog (@"y origin: %f", testRect.origin.y); // -3
NSLog (@"width: %f", testRect.size.width); // 10
NSLog (@"height: %f", testRect.size.height); // 15
```

Note that you first access the `origin`, and then its coordinates, and similarly for the `size`.

The constant `NSZeroRect` represents a rect with `NSZeroPoint` `origin` and `NSZeroSize` `size`.

1.4 Geometry Functions

The GNUstep base library provides functions and macros to do all the most common geometry operations on `NSPoint`s, `NSSize`s and `NSRect`s (such as determining if a point is contained in a rectangle, computing the intersection of two rectangles, etc). It would be off topic to discuss them here; please refer to the base library documentation whenever you need to do some of these common geometrical operations.

2 Adding a Window to your Application

2.1 Window Attributes

Windows are represented in GNUstep by `NSWindow` objects. We are mainly interested in two attributes of a `NSWindow` object:

1. its *content rect*, which is a `NSRect` describing the rectangle covered by the internal area of the window; i.e., without all the decorations added by the window manager (title bar, border, etc);
2. its *style mask*, which is an `unsigned int` describing the decorations of the window. A zero style mask is also known as a `NSBorderlessWindowMask`, and means that the window has no decorations at all. If the window needs to have decorations, you need to set its style mask to a combination (`OR`) of one or more of the following constants:
   - `NSTitledWindowMask` = the window has a title bar;
   - `NSClosableWindowMask` = the window can be closed by the user (it has a close button);
   - `NSMiniaturizableWindowMask` = the window can be miniaturized by the user (it has a miniaturize button);
   - `NSResizableWindowMask` = the window can be resized by the user.

   For example, a window which has a title bar, is closable and miniaturizable will have a style mask of:

   ```
   unsigned int windowMask = NSTitledWindowMask
                             | NSClosableWindowMask
                             | NSMiniaturizableWindowMask;
   ```

   where `|` is the C operator for the logical OR.

While you might freely change the content rect of your window after the window has been created, you can only set the style mask when the window is first created. You can not change the style mask of the window after creation.

2.2 Creating a Window

We are now ready to show an example of creating a window:

```
NSWindow *myWindow;
NSRect rect = NSMakeRect (100, 100, 200, 200);
unsigned int styleMask = NSTitledWindowMask
                         | NSMiniaturizableWindowMask;

myWindow = [NSWindow alloc];
myWindow = [myWindow initWithContentRect: rect
                     styleMask: styleMask
                     backing: NSBackingStoreBuffered
                     defer: NO];
```

(Please ignore the `backing:` and `defer:` arguments for now. The values given in the example are appropriate in most cases). This window has an initial content rect which has its origin (its lower left corner) positioned in the point of coordinates (100, 100) in the screen coordinates. In GNUstep, all coordinates system are by default Cartesian and the origin is always in the lower left corner (this is only the default, because inside a GNUstep window you can then flip, rotate and (more generally) transform coordinates at your wish). The window has a title bar, and can be miniaturized by the user, but not resized or closed.

2.3 Setting the title of a window

To set the title of the window `myWindow`, you simply do something like:

```
[myWindow setTitle: @"This is a test window"];
```

2.4 Ordering Front a window

Creating a window does not make it visible. The window is ready to be used, but it is not visible till you invoke the `orderFront:` method, as in:

```
[myWindow orderFront: nil];
```

Note that `orderFront:` takes an object argument so that it can be invoked as an *action* from a button or a menu item, but the argument is usually ignored, which is why we simply use `nil` for it.

If you want your window to be ordered front and to get the keyboard focus, you do:

```
[myWindow makeKeyAndOrderFront: nil];
```

2.5 Integrating the window with your application

We now want to show a very simple example of an application creating a single window. Our application will be useless if its only window is not visible; for this reason, we do not add a close button to the window (following the *NEXTSTEP* tradition, the user needs to select *Quit* from the main menu to quit the application. To get the main menu, the user typically clicks the right button of his mouse on one of the windows).

We can create the window in the same method

```
- (void) applicationWillFinishLaunching: (NSNotification *)not;
```

where we created the menu. This method contains initialization code to be run *before* the application is launched.

But, we can't order front (ie, make visible) windows before `NSApp` has become active (which happens when the application is 'launched'), so that ordering front our window in `applicationWillFinishLaunching:` would not work, because this method is called *before* the application finished launching.

To get around this problem, we implement in our application delegate also another method, called:

```
- (void) applicationDidFinishLaunching: (NSNotification *)not;
```

This method (if implemented by the application's delegate) is invoked by the `NSApp` just after it finished launching. We order front our window in this method.

2.5.1 The organization of start-up code

The way we have organized the start-up of our application is as follows:

```
- (void) applicationWillFinishLaunching: (NSNotification *)not;
```

contains code for the initialization process of the application. This usually means creating the application's main objects; in our case, the menu and the only application's window. You can't (and shouldn't) display any window in this method.

Instead, the method

```
- (void) applicationDidFinishLaunching: (NSNotification *)not;
```

contains a sequence of user-visible actions to be taken just after the application became active: typically, popping up a window or a panel to the user.

It should be mentioned that this is not the only possible way of organizing the start-up of an application.

2.6 Source Code

Here is the source code of our new application:

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

@interface MyDelegate : NSObject
{
  NSWindow *myWindow;
}
- (void) createMenu;
- (void) createWindow;
- (void) applicationWillFinishLaunching: (NSNotification *)not;
- (void) applicationDidFinishLaunching: (NSNotification *)not;
@end

@implementation MyDelegate : NSObject
- (void) dealloc
{
  RELEASE (myWindow);
}

- (void) createMenu
{
  NSMenu *menu;

  menu = AUTORELEASE ([NSMenu new]);

  [menu addItemWithTitle: @"Quit"
        action: @selector (terminate:)
        keyEquivalent: @"q"];
```

```
  [NSApp setMainMenu: menu];
}

- (void) createWindow
{
  NSRect rect = NSMakeRect (100, 100, 200, 200);
  unsigned int styleMask = NSTitledWindowMask
                           | NSMiniaturizableWindowMask;


  myWindow = [NSWindow alloc];
  myWindow = [myWindow initWithContentRect: rect
                       styleMask: styleMask
                       backing: NSBackingStoreBuffered
                       defer: NO];
  [myWindow setTitle: @"This is a test window"];
}

- (void) applicationWillFinishLaunching: (NSNotification *)not
{
  [self createMenu];
  [self createWindow];
}

- (void) applicationDidFinishLaunching: (NSNotification *)not;
{
  [myWindow makeKeyAndOrderFront: nil];
}
@end

int main (int argc, const char **argv)
{
  [NSApplication sharedApplication];
  [NSApp setDelegate: [MyDelegate new]];

  return NSApplicationMain (argc, argv);
}
```

To make the code easier to read and to understand, we have moved all the code creating the menu into a `createMenu` method, and the code creating the window into a `createWindow` method.

We have also implemented the `dealloc` method, which is quite useless in this case, because we create only one `MyDelegate` object, which is only released when the application quits (there is few interest in releasing memory when the application is quitting, since all memory is going to be released anyway). But it is good programming practice to always implement `dealloc`; this method should release all the resources that the object was using. In this case, we only need to release the window object. Please refer to the Memory Management tutorial [yet to be written *grin*] for more information on the `dealloc` method.

The `GNUmakefile` is the usual one:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = WindowApp
WindowApp_OBJC_FILES = MyApp.m

include $(GNUSTEP_MAKEFILES)/application.make
```

3 Adding a Button in the Window

We are now going to add a button inside our window. But first, we need to prepare the ground with a very short introduction to the wonderful `NSView` class.

3.1 A Quick Introduction to Views

`NSView` is a class whose objects represent a rectangular area in a window. Each `NSView` has its own internal coordinate system; `NSView` provides a complete framework for managing the rectangle represented by the view, its coordinate system, and for drawing and getting user events inside the view's rectangle.

Any object which can be displayed (and/or can accept user events) inside a window (such as a button, a text field, a scroll view, a slider, etc) is actually an instance of a subclass of `NSView`. The specific subclass (for example, `NSButton`) implements a specific way of drawing and managing user events inside the view's rectangle. An unsubclassed `NSView` by default draws nothing and reacts to no user events, even if it contains all the machinery (ready for subclasses to use) to do both.

When you create a window, the library automatically creates an `NSView` covering the whole content area of the window. This view is called the window's *content view*. You can replace the default content view (which is transparent and reacts to no user events) with an `NSView` of your choice (which can be an instance of a subclass of `NSView`, such as a `NSButton` or a `NSScrollView`), but you should not change the rectangle it represents, which must always cover the whole window.

If you want to have a view which covers only a part of a window, you need to add it as a *subview* of the content view. We will explore subviews in a forth-coming tutorial; in our first example, we will go for the simplest possible solution avoiding use of subviews: we will create a single `NSButton`, make a window to fit exactly our button, and then replace the default window's content view with our button. In this way we'll get a window containing exactly our button. In the next tutorial, we'll learn how to enclose views one over the other in a subview tree, so that we can put more things in a single window.

3.2 Creating a Button

Creating a button is quite easy:

```
NSButton *myButton;

myButton = AUTORELEASE ([NSButton new]);
[myButton setTitle: @"Print Hello!"];
```

```
[myButton sizeToFit];
```

setTitle: sets the title (string) to be displayed on the button; sizeToFit resizes the button so that it fits the title it is displaying.

Next, we need to set target and action of the button:

```
[myButton setTarget: self];
[myButton setAction: @selector (printHello:)];
```

(self will refer to our custom object)

3.3 Putting the button in the window

We now need to modify our code to create the window so that it creates a window exactly of the right size to contain our button. First, we need to get the size of the button:

```
NSSize buttonSize;

buttonSize = [myButton frame].size;
```

frame is a method inherited by NSView which returns the rectangle enclosing the button (or, more generally, the rectangle enclosing/represented by an NSView). The origin of this rectangle is meaningless at this point; it is the size (which was set by sizeToFit) what we want.

Then, we choose for the window a content rect with the size of the button, and with an origin of our choice; (100, 100) in this example:

```
NSRect rect;

rect = NSMakeRect (100, 100,
                   buttonSize.width,
                   buttonSize.height);
```

At this point, we just create the window as before, but using this rectangle (full code below). The last thing to do is then replacing the default window content view with our button:

```
[myWindow setContentView: myButton];
```

3.4 Source Code

We are ready to show the whole source code (the GNUmakefile is the usual one):

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>

@interface MyDelegate : NSObject
{
  NSWindow *myWindow;
}
- (void) printHello: (id)sender;
- (void) createMenu;
```

```
- (void) createWindow;
- (void) applicationWillFinishLaunching: (NSNotification *)not;
- (void) applicationDidFinishLaunching: (NSNotification *)not;
@end

@implementation MyDelegate : NSObject
- (void) dealloc
{
  RELEASE (myWindow);
}

- (void) printHello: (id)sender
{
  printf ("Hello!\n");
}

- (void) createMenu
{
  NSMenu *menu;

  menu = AUTORELEASE ([NSMenu new]);

  [menu addItemWithTitle: @"Quit"
        action: @selector (terminate:)
        keyEquivalent: @"q"];

  [NSApp setMainMenu: menu];
}

- (void) createWindow
{
  NSRect rect;
  unsigned int styleMask = NSTitledWindowMask
                           | NSMiniaturizableWindowMask;
  NSButton *myButton;
  NSSize buttonSize;

  myButton = AUTORELEASE ([NSButton new]);
  [myButton setTitle: @"Print Hello!"];
  [myButton sizeToFit];
  [myButton setTarget: self];
  [myButton setAction: @selector (printHello:)];

  buttonSize = [myButton frame].size;
  rect = NSMakeRect (100, 100,
                     buttonSize.width,
                     buttonSize.height);

  myWindow = [NSWindow alloc];
  myWindow = [myWindow initWithContentRect: rect
```

```
                       styleMask: styleMask
                       backing: NSBackingStoreBuffered
                       defer: NO];
  [myWindow setTitle: @"This is a test window"];
  [myWindow setContentView: myButton];
}

- (void) applicationWillFinishLaunching: (NSNotification *)not
{
  [self createMenu];
  [self createWindow];
}

- (void) applicationDidFinishLaunching: (NSNotification *)not;
{
  [myWindow makeKeyAndOrderFront: nil];
}
@end

int main (int argc, const char **argv)
{
  [NSApplication sharedApplication];
  [NSApp setDelegate: [MyDelegate new]];

  return NSApplicationMain (argc, argv);
}
```