POSTSCRIPT®

THE DISPLAY POSTSCRIPT® SYSTEM: NeXT INTERFACE BUILDER and the HELLO WORLD APPLICATION

Technical Note #5051

March 6, 1990
PostScript® Developer Support Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5051

THE DISPLAY POSTSCRIPT® SYSTEM: NeXT INTERFACE BUILDER and the HELLO WORLD APPLICATION

POSTSCRIPT®

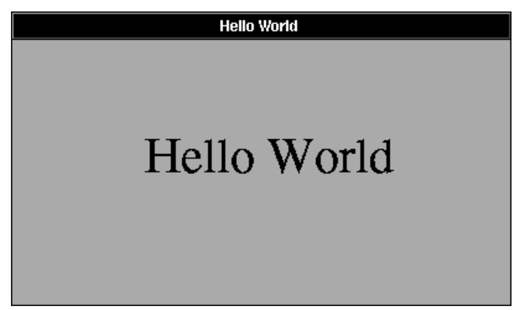
Technical Note #5051

March 6, 1990
PostScript ® Developer Support Group
(415) 961-4111

1. INTRODUCTION

HelloWorld is a simple application with two purposes — to show how to use Interface Builder to create an application and to introduce drawing in a view.

HelloWorld has a menu and a window. There are four buttons on the menu - **Hello World**, **Clear**, **Hide** and **Quit**. When **Hello World** is selected, the text, “Hello World”, is displayed in the window. When **Clear** is selected, the window is cleared. **Hide** hides the application and **Quit** quits the application.



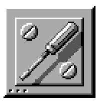
2. INTERFACE BUILDER

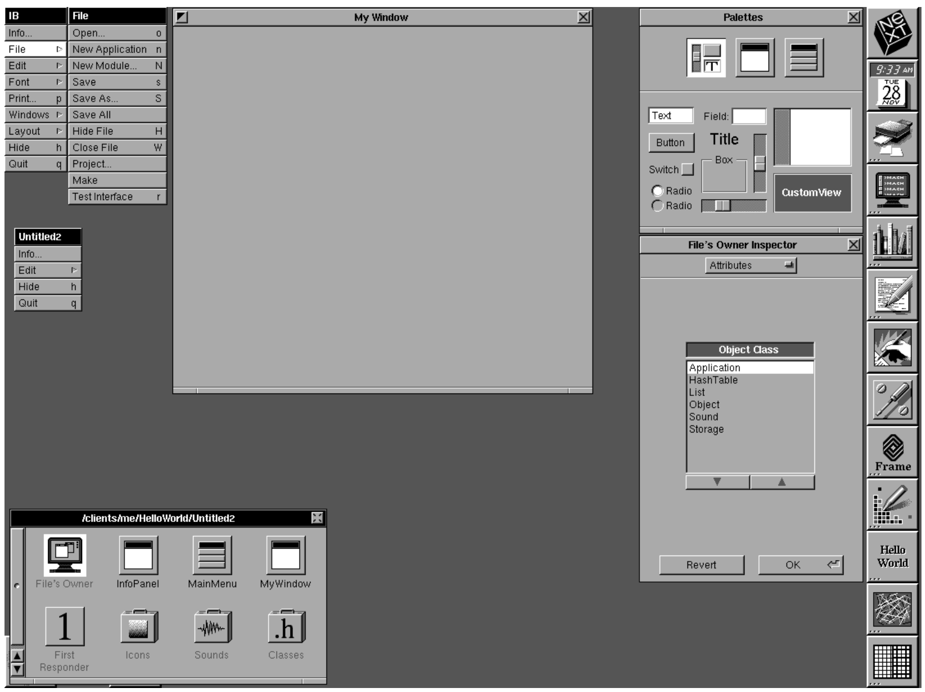
Interface Builder is an application supplied with the NeXT™ computer that allows developers to quickly create a user interface for an application. Windows, buttons, sliders and other related Application Kit objects can be graphically placed on the screen and incorporated into an application. The attributes of an object can be edited. An example is giving a window a title and a buffered backing instead of a retained backing. Certain connections can be made between objects so that, for example, selecting a menu item will cause a window to appear or sliding a slider will change the value in a text field.

Interface Builder creates and stores the interface instructions in a **.nib** file (which stands for NeXT Interface Builder). This file is then linked to the application.

3. HELLOWORLD APPLICATION

This section and the sections that follow will use Interface Builder to create the user interface for **HelloWorld**. The first step is to launch Interface Builder and select the **New Application** menu option which appears under **File** option in the main menu. A menu, a window and several panels should appear.





3.1 Nib File

This newly created nib file should be saved into a new directory. Select **Save** and in the **Name** entry field enter a new directory name followed by a slash and a file name — **HelloWorld/HelloWorld.nib**. Then click **OK** or type return. (A panel will appear confirming the creation of the path **~/HelloWorld** if it does not exist. Answer **Yes**.) Now select the **Project** menu option, again under the **File** option. The **Project Inspector** panel will appear with a message asking if a project file should be created in the directory. Again answer **Yes**. This step creates the **Makefile**, **HelloWorld.iconheader**, **HelloWorld_main.m** and **IB.proj** files.

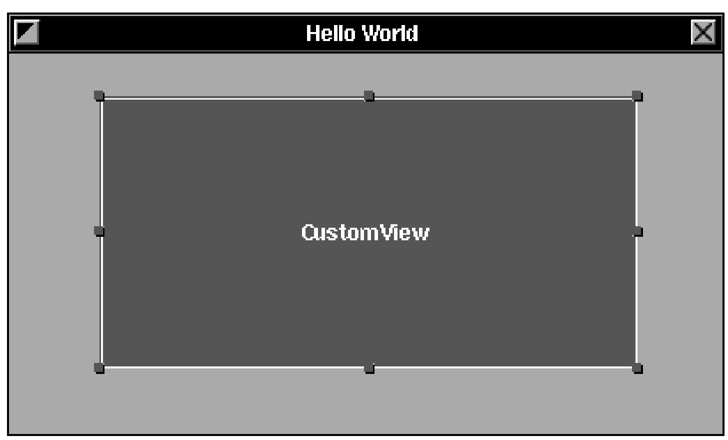
3.2 Menu

Now that some of the project administration is out of the way, let's move to the application itself. The first step is to modify the menu. Delete the **Info...** and **Edit** menu items. Selecting and then cutting or clicking the **Delete** in the Interface Builder **Edit** submenu will accomplish this. Next, additional menu items should be placed in the menu. Click the rightmost of the three icons at the top of the Palettes panel (the menu icon). Select and drag an **Item** bar from the palette. Place it in the menu and then release. The **Item** bar should appear in the menu resized to match the existing items. Add another by repeating the process or by selecting the existing **Item** bar in the menu and copying/pasting.

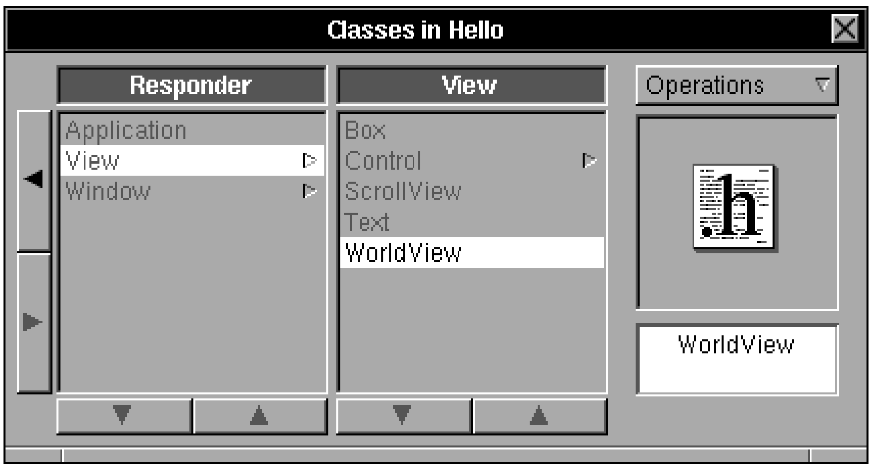
Change the text in the top item bar to **Hello World** by double clicking in the cell, selecting the existing text and typing “Hello World”. (A return or movement to another cell will cause the menu to resize to the correct size and show all the text.) Change the text in the second item bar to **Clear** by double clicking, selecting and then typing “Clear”. The **Hide** and **Quit** menu items are fine the way they are.

3.3 WorldView

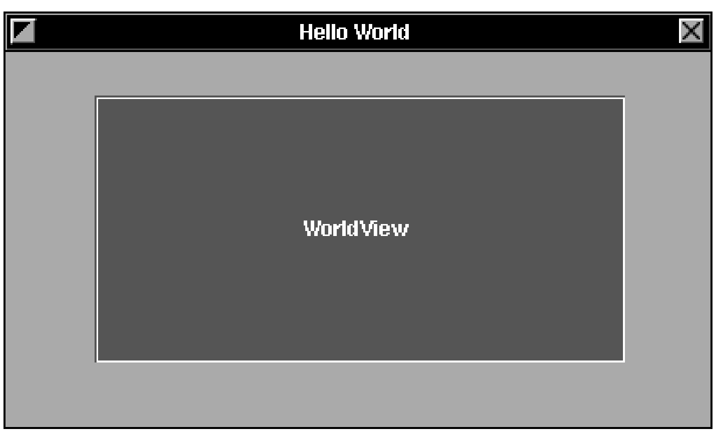
Next, a view should be placed in the window. Resize the window to the desired size and move it to where you want it to first appear when the application launches. (The title of the window can be changed from **MyWindow** to **Hello World** through the **Attributes Inspector**.) Change the **Palette** panel to sliders/buttons/text fields and then select and drag the **CustomView** object into the window. Enlarge the **CustomView** in the window. It does not have to enclose the entire window but should be large enough to contain the “Hello World” message. When the application runs, the view will not appear like the **CustomView** in Interface Builder. The background of the view will be drawn to match the background of the window.



The next step is to turn this object into a subclass of **View** so that drawing instructions can be added. Double click the **Classes** icon in the **Objects** panel at the lower left. A **Classes in Hello** panel should appear. Scroll through and select the **View** class. The subclasses of **View** should be **Box**, **Control**, **ScrollView** and **Text**. None of these should be selected. Move to the **Operations** button and select **Subclass**. A subclass named **Subclass1** should appear in the panel next to **View**. Change the name of **Subclass1** to **WorldView** in the text entry under the icon. (**WorldView** should be a subclass of view and not a subclass of **Box**, **Control**, **ScrollView** or **Text**.)

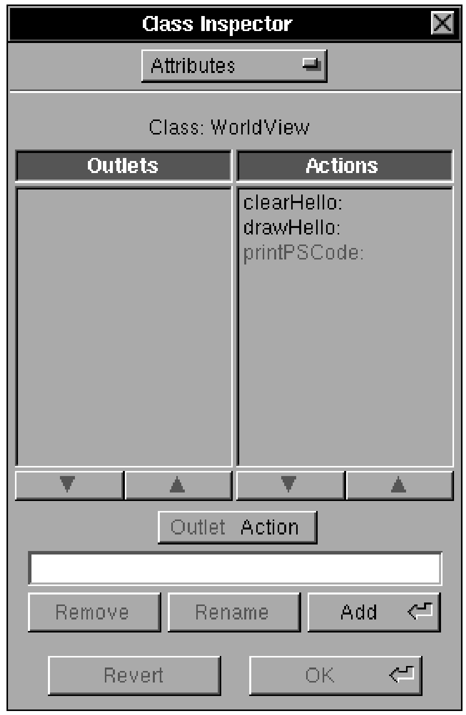


Once the **WorldView** subclass has been created, the **CustomView** object in the window needs to be changed from a **View** class to a **WorldView** class. Move to the **Inspector** panel and change it from **Project Inspector** to **Attributes Inspector** (the button at the top will do this). Then select the **CustomView** in the window. The **Attributes** panel should show **CustomView** as a **View** class. Change the class from **View** to **WorldView** and select **OK** or type a return. The **CustomView** in the window should now be renamed **WorldView**.



3.4 Method Names

Our next step is to make the connections between the menu items and the methods that will be in the **WorldView** class. In order to form these connections graphically, we must first define the methods of **WorldView** within Interface Builder. Select **WorldView** and then change the **Inspector** panel from an **Attributes Inspector** to a **Class Inspector**. An outlet/ action panel should appear with the single action **printPSCode:** in gray.



Method names for the **WorldView** object will be entered as actions and will appear next to the **printPSCode** method. Make sure that **Action** is selected in the **Outlet/Action** button approximately 3/4 of the way down the window. Enter the method name **drawHello** and return (or select the **Add** button). Enter the method name **clearHello** and return (or select the **Add** button). (A colon after the name is not necessary since Interface Builder appends one if it does not appear.) Now that the method names in the **WorldView** class are defined within Interface Builder, we can form the connection between a menu cell and a method name.

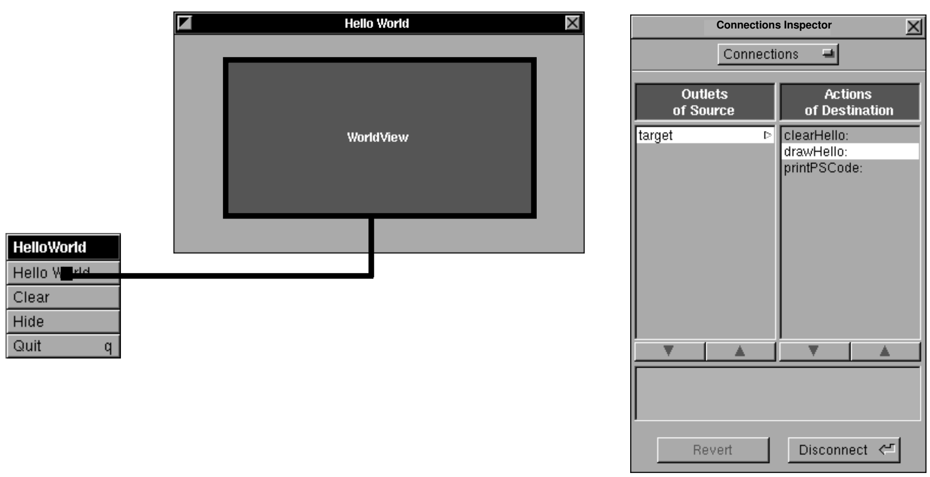
3.5 Connections

Making a connection between objects simply means that a message is placed in the nib file that is sent between the objects when the appropriate event is received by the sender at run-time. In this case, messages will be sent to the **WorldView** object whenever the **Hello World** or **Clear** items in the menu have been selected.

To make the first connection, hold the control key down while selecting the **Hello World** menu cell. A small black square should appear in the center of the cell with a black line attached. Continue holding the mouse down and move to the **WorldView** object. As soon as the mouse nears **WorldView**, a black line should surround the **WorldView** and connect to the existing black line. Release the mouse — the black lines and square should remain. This action will cause the list of actions or methods within **WorldView** to appear. We can then specify which action should be messaged when the menu cell is selected.

The **Inspector** panel should automatically change to a **Connections Inspector** with the **drawHello**, **clearHello** and **printPSCode** appearing in the **Actions of Destinations** view. Select the **drawHello** method and then select the **Connect** button at the bottom. This will form the connection between the menu cell and the action so that a message will be sent to the **drawHello** method in **WorldView** when this menu cell is selected. Repeat the same steps with the **Clear** menu cell, selecting the **clearHello** action instead of the **drawHello** action.

Note: *Make sure that the* ***Connect*** *button is selected or that a return is typed after choosing the action or else the connection will not take effect.*



3.6 Unparsing the WorldView Class

The next step is to have Interface Builder create initial **WorldView.h** and **WorldView.m** files. Move to the **Classes in Hello** panel and select the **WorldView** class. Move to the **Operations** button and select the **Unparse** option. Answer **Yes** to the two prompts. The first is to confirm the creation of the files and the second is to confirm the inclusion of the files in the makefile. Now we can save the nib file we have created, hide or quit Interface Builder and complete the **WorldView** files.

3.7 WorldView Files

Launch the Edit application or another editor and open up the files that Interface Builder has created, **WorldView.h** and **WorldView.m**. The files should appear as below. Objective-C statements need to be inserted in the files in order to make **WorldView** perform its intended operations.

The header file:

```
WorldView.h

/* Generated by Interface Builder */

#import <appkit/View.h>

@interface WorldView:View
{
}

- drawHello:sender;
- clearHello:sender;

@end
```

and the implementation file:

```
WorldView.m

/* Generated by Interface Builder */

#import "WorldView.h"

@implementation WorldView

- drawHello:sender
{
    return self;
}

- clearHello:sender
{
    return self;
}

@end
```


WorldView.h

The first step is to modify the interface file, **WorldView.h**. An instance variable should be added to keep track of whether we should display “Hello World” or not. Add a **BOOL** value and call it **DrawHello**. Next we need to add a method to the interface file. Actually, we are not really adding a *new* method to **WorldView**. We are *overriding* an existing method inherited from the **View** class. The method to add is **drawSelf::** which is automatically messaged whenever the view is displayed.

```
- drawSelf:(NXRect *)r :(int) count;
```

With the addition of comments, the interface file should now look like the following. Save it and move to **WorldView.m**.

```
/*
 * WorldView.h
 *
 * The instance variable, DrawHello, is toggled between YES and NO
 * and indicates whether the "Hello World" message should be
 * displayed in the view.
 */

#import <appkit/View.h>

@interface WorldView:View
{
  BOOL DrawHello;
}

- drawHello:sender;
- clearHello:sender;
- drawSelf:(NXRect *)r :(int) count;

@end
```

WorldView.m

In **WorldView.m**, the two lines below in bold should be added to the method, **drawHello**. The lines should be placed before the return statement.

```
- drawHello:sender
{
    DrawHello = YES;
    [self display];
    return self;
}
```

The two lines in bold below should be added to the method, **clearHello**. The lines should again be placed before the return statement.

```
- clearHello:sender
{
    DrawHello = NO;
    [self display];
    return self;
}
```

The first line of each method toggles the boolean instance variable, **DrawHello**, between YES and NO – YES for drawing "Hello World" and NO for not drawing "Hello World". The next line messages **self** (the **WorldView** object's own id) and tells it to perform the method, **display**. The **display** method is a method defined in **View**, a method that **WorldView** inherits. Drawing in a view is accomplished by placing the drawing instructions in the **drawSelf::** method, overriding the default **drawSelf::** method from **View**. This method is not messaged directly; instead, **display** is messaged which in turn will message **drawSelf::**.

Since we will be drawing in **WorldView**, we need to override the **drawSelf::** method. Insert the method listed below after the **clearHello** method. The drawing consists of clearing the previous drawing by filling the view with light gray and displaying "Hello World" if **DrawHello** is YES.

```
- drawSelf:(NXRect *) r: (int) count
{
  PSsetgray(NX_LTGRAY);
  PSrectfill(
    bounds.origin.x, bounds.origin.y,
    bounds.size.width, bounds.size.height);

  if (DrawHello)
  {
    PSsetgray(NX_BLACK);
    PSmoveto(50.0, 70.0);
    PSselectfont("Times-Roman", 40.0);
    PSshow("Hello World");
  }

  return self;
}
```

Drawing will be done using the single operator calls in the Client Library. These calls are fine for simple drawing like that found in this example, but more complex drawing should use the advantages that the *pswrap* translator provides. Clearing any previous drawing consists of two operations – setting the gray and filling the view. The **bounds** structure in the **PSrectfill( )** call is an instance variable found in the **View** class that gives the origin, width and height of the view. **PSrectfill( )** fills the view with the current color in the PostScript graphics state, light gray. The gray level chosen matches the background of the window so it doesn't really matter if the view is smaller than the window.

The next group of lines shows the text "Hello World" in the view if **DrawHello** equals YES. The gray level will be set to black and a currentpoint will be specified in the PostScript graphics state. The PostScript operator, **selectfont**, establishes the current font in the graphics state – 40 point, Times-Roman. The **show** operator displays the string "Hello World" in the current font at the current point in the current color.

Note: *The point (50.0, 70.0) is relative to the lower left corner of the view placed within the window in Interface Builder. A different point will more than likely be necessary for a differently sized or differently placed view.*

Include the **#import <dpsclient/wraps.h>** line, add a few comments if desired and then save the file. The complete file can be found below.

```
/*
 * WorldView.m
 *
 * WorldView is a subclass of View. It simply displays and clears
 * a message in the view. The one lesson this example shows is the
 * modification of "drawSelf::" and the call to"display". The method,
 * "drawSelf::", should not be called directly.
 *
 * Rather, "display" should be called. Why? Because display message
 * performs some necessary overhead such as bringing the View into
 * focus by constructing a clipping path around its frame rectangle
 * and making its coordinate system the current coordinate system
 * for the application. The display method then messages drawSelf::.
 * These steps are repeated for each of the View's subviews.
 *
 * Note: Any instance variables that need to be initialized before
 * the first display should be done in the "new" or "newFrame:"
 * method. In this case, DrawHello is already a null value at its
 * creation so no initialization has to be performed.  */

#import "WorldView.h"
#import <dpsclient/wraps.h>

@implementation WorldView

- drawHello:sender
{
    DrawHello = YES;
    [self display];
    return self;
}

- clearHello:sender
{
    DrawHello = NO;
    [self display];
    return self;
}

/* The first two lines clear the view. The remainder display the
 * message in the view. */
- drawSelf:(NXRect *) r: (int) count
{
  PSsetgray(NX_LTGRAY);
  PSrectfill(bounds.origin.x, bounds.origin.y,
      bounds.size.width, bounds.size.height);

  if (DrawHello)
  {
    PSsetgray(NX_BLACK);
    PSmoveto(50.0, 70.0);
    PSselectfont("Times-Roman", 40.0);
    PSshow("Hello World");
  }
  return self;
}
@end
```

3.8 Making and Running

The **make** command can be specified from Interface Builder (in the **File** submenu) or from the shell. Typing in **make** in the shell will make the executable file **HelloWorld**. (The application name can be specified in the **Project Inspector**. It takes by default the name of the nib file.). Once the make has completed successfully (in other words, once the compiler errors have been resolved), the application can be launched. (The make option in Interface Builder creates a debug version of **HelloWorld**, **HelloWorld.debug**, for use with the gdb debugger. It can also be executed outside the debugger.)

Typing **HelloWorld** in the Shell or double clicking the file in the Workspace Manager will start the application. The application should work as described in the beginning of this note.

3.9 Common Problems

- *Selecting a menu item does nothing*.

  Make sure the menu item is connected to the appropriate method within Interface Builder. A return has to be typed or the Enter button selected after the method has been selected in order for the connection to take effect. Connecting the objects and then selecting the method is not enough. Also, make sure the two methods entered in the Class Inspector have the same spelling as those in the WorldView.h and WorldView.m files.

- *The string "Hello World" is either clipped or does not appear.*

  The point, (50.0,70.0), may not position the text in the correct spot. Continue reading and then try making some adjustments to the coordinate point.

4. TECHNICAL NOTES

4.1 View

Aside from introducing Interface Builder, the primary purpose of this exercise is to introduce drawing in a view. A **View** is an object in the Application Kit that provides a structure for drawing on the screen (and for handling mouse and keyboard events). All the drawing that is done on the screen is done using **View** objects. The **Windows**, **Panels**, **Buttons** and **TextFields** either contain **View** objects or are subclasses of **View**. **Windows** and **Panels** contain **View** objects. **Buttons** and **TextFields** are subclasses of **Views**.

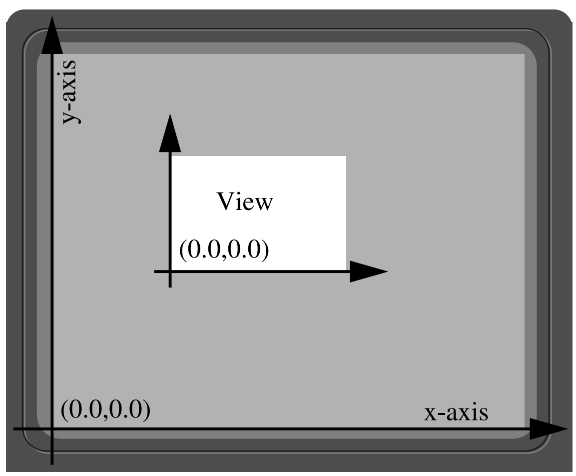
4.2 Window

The **Window** object in **HelloWorld** contains two views, a *frame view* and a *content view*. The frame view consists of the border, the title bar and the resize bar. The content view consists of the area inside the border and bars. The frame view is *private* to all but the window and should not be altered by an application. The content view is *public*. It can contain other views and can even be replaced by another view. In **HelloWorld**, the content view is kept as a subview of the window and **WorldView** is made a subview of the content view.

4.3 Coordinate System

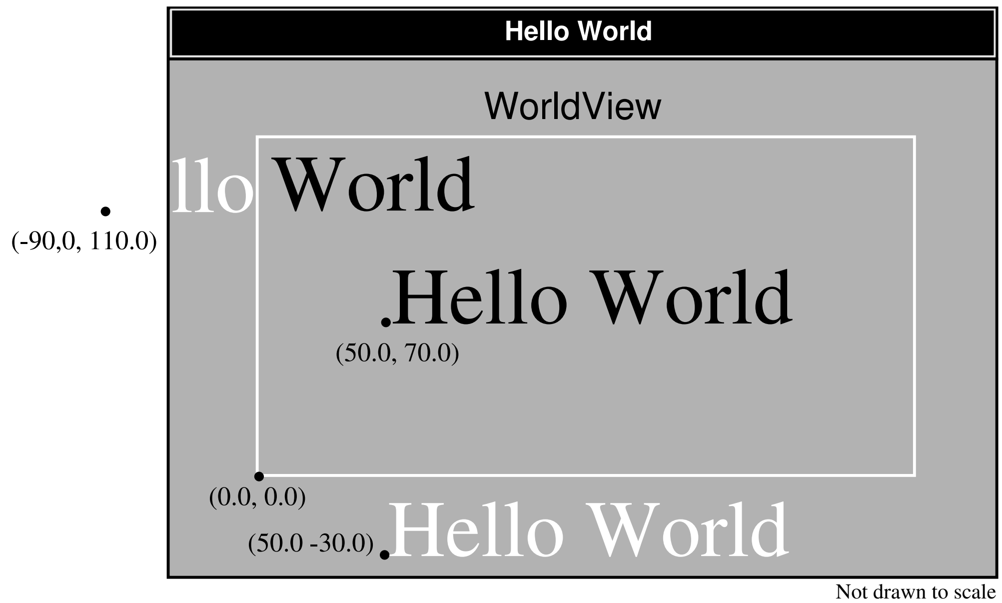
The Display PostScript system uses the Cartesian coordinate system to specify location on the screen. The origin is originally at the lower left of the screen. The positive Y axis is to the top and the positive X axis is to the right. Each point is equal to approximately 1/92 of an inch on the NeXT computer. An important feature to note is that the origin can be moved to another location. As a result, instructions that are used in one location can be used in another simply by moving or *translating* the origin of the coordinate system. For example, consider the drawing of two identical buttons. Only one set of PostScript language instructions needs to be used to describe the look of the buttons. Their different positions can be accomplished by translating the origin.

The same concept applies to views as well. A View has a width and a height and a position on the screen. It also has its own coordinate system so that drawing within the view is relative to the view only and not to its position on the screen or within a window. The origin for the view is its lower left corner. The instance variable, **bounds**, contains the origin of the view which is usually the point (0.0, 0.0) and the width and height of the view. It is these values, **bounds.origin.x**, **bounds.origin.y**, **bounds.size.width** and **bounds.size.height**, that are passed to **PSrectfill( )**.



The display of **Hello World** includes a call to **PSmoveto( )** with the point (50.0,70.0). This point is roughly an inch up and half an inch to the right of the view's bottom left corner. Note that the origin when we are drawing is the lower left of the *view* and not the lower left of the *window*. By default, a view sets a clip path around the dimensions of its frame. Anything that is outside the frame does not get shown. As a result, moving to a point of

(50.0, -30.0) and performing the **PSshow( )**would not display any text. Likewise, the point (-90.0, 110.0) would show only a portion of the text. In the diagram below, the text in white is what would not be shown due to the clipping region of the view.



4.4 Messaging Display

To draw in a view, the **drawSelf::** method in the view subclass must be overridden. The actual execution of the drawing, however, must be accomplished by sending a message to the **display** method and not **drawSelf::**. In the **Hello World** example, both **drawHello** and **clearHello** message the **display** method. This method is found in the **View** class and performs necessary overhead for drawing in the view. The **display** method first brings the view into focus which means that a clipping path is created around its frame rectangle and the view's coordinate system is installed as the current coordinate system in the PostScript language graphic state. (The coordinate system is covered in more detail in Technical Note #5053.) After the view has been brought into focus, **display** messages the **drawSelf::** method. Then, **display** messages are sent to all of the subviews. Any view that has its subviews do all the drawing does not need to override **drawSelf::**. The **display** message of the superview will message all its subviews.

5. SUMMARY

Almost all drawing on the NeXT computer is performed within a subclass of a **View** class. The methods provided in the **View** class handle much of the overhead for drawing. The actual drawing should be placed in the **drawSelf::** method of the subclass. This method should not be called directly. Instead, **display** should be messaged which in turn will message **drawSelf::**.

Only the most basic issues involved in drawing within a view have been touched in this note. The material here is sufficient to understand the concept of drawing in a view. The NeXT documentation, however, contains more detailed information.

