

CSE212 LAB ASSIGNMENT

Week of March 31, 2003

WINDOWS PROGRAMMING II

This assignment extends last week's lab assignment by creating a slot machine application. In addition to buttons, we will see how to use text boxes to get data, use static text boxes as labels, and illustrate the use of bitmaps.

It is notable that this battery of lab assignments makes use of Win32 API programming and **not** Microsoft Foundation Classes (MFC). This is intentional so that you get to see what is going on "behind the scenes" at a lower level than you would have to deal with otherwise. Using the MFC provides a somewhat simpler interface for Windows controls. *This lab is the second of three covering Windows programming (GUI and networking).*

Step by step instructions:

1. Read this handout before starting the lab. Recommended additional resources are available from the following links:
 - a) From a Western Illinois University course on Windows GUI programming; in particular you may want to look at the link titled *helpful table of controls*, which can be found near the end of the **Lecture Notes** section:
<http://www.wiu.edu/users/mflll/CS412g/CS412g.html>
 - b) A variety of tutorials and programs are available from this site, but this one in particular may be helpful:
<http://www.cplusplus.com/src/w32sk.zip>
2. Download the files "slots.cpp", "resource.h", and "gui_project.rc" from the course website to your personal space.
3. Create a new, empty Win32 application project named "slots" in the location of your choosing.
4. Add the file "slots.cpp" to the source code files, "resource.h" to the header files, and "gui_project.rc" to the resources files. The application should compile and run as is, although we are going to add some features. The labels have been written for you, as have some edit boxes and bitmap resources. For more information on possible settings, look up STATIC, EDIT, and BUTTON controls and constants in the help files.
5. First, we are going to add a message box that displays whether the player wins, loses, or pushes (neither wins nor loses) the bet.
 - a) Add a new "Edit Box" control called `hwndEditWinLose` at position `x=255`, `y=165` of length=120 and height=20. This is similar to the `hwndEditInstr` edit box that displays instructions for the player and should appear just above it on the GUI. Note that you need to both declare the variable and add the control to the `WinMain()` function. Do not include an initial text value.
 - b) Add three new variables of type `char*` called `msgWin`, `msgLose`, and `msgPush` that will be used to signify a winning, losing, or no money exchanged

- result respectively for a round of play. Be aware that the edit box can only display a limited number of characters.
- c) In the `enableControls()` function, add code that will display the messages you defined in part 5b in the win/lose edit box. Make the displayed message dependent on the value of the `amount` parameter. Use the `SetWindowText()` function, passing the edit box as the 1st parameter and your message as the second.
 - d) Build and run the application to make sure it works.
6. Next, we will add an additional bitmap (a stein of beer) to the selection for display. The bitmaps are stored in an array called `bmps[]` for easy access.
 - a) Add a variable `stein` of type `HBITMAP` and increase the value of `NUM_BMPS` by one to make it 9.
 - b) In the `InitBitmapResources()` function, add code to load the stein bitmap and put it in the bitmap array at index 8.
 - c) Finally, adjust the `calculateTwoSlots()` function so that whenever two steins are displayed the player wins double their bet. In addition, add code so that if a stein is displayed in either location the player wins \$50. (Of course, this is all a ploy by the casino to get the patrons to drink more and spend all their money. Either that or someone in charge figures that you can't lose with steins of beer.)
 - d) Build and run the application to make sure it works.
 7. Next, we will add an additional "slot" so that there are three bitmaps displayed instead of two. This entails not only adding a bitmap to the display where necessary, but also changing the winnings calculations.
 - a) Increase the value of `NUM_SLOTS` by one to 3.
 - b) Add code that will display the additional bitmap at position 250, 20 by setting the `loc[2]` variable appropriately (which contains an index of the bitmap array) and using the `ShowBitmapResource(HBITMAP, HDC, x, y)` function. This must be done in the `MainWinProc()`, `animate()`, and `WinMain()` functions. Look for similar code and use it as an example of your changes.
 - c) Since the value of `NUM_SLOTS` is now 3, the `calculateBankroll()` function will now call the `calculateThreeSlots()` function. Change the code so that if three steins come up, the player wins double their bet (some code is already there). Also add code so that if two steins come up, the player wins \$200, no matter what their bet was.
 - d) Build and run the application to make sure it works.
 8. Finally, we will assign some probabilities to each bitmap. The code to do this has been written for you.
 - a) At the end of the `InitBitmapResources()` function, add a function call to `assignProbs()`. This will fill the `probs[]` array with values. The indices match those of the `bmps[]` array; for instance, the value you put in index 8 controls the probability that a stein will come up.
 - b) Modify the `animate()` function so that before returning (that is, after the outer for loop), the values for `loc[0]`, `loc[1]`, and `loc[2]` are set using a call to `GetBitmapIndexUsingProbability()`. Also add code to display the new bitmaps after the probabilistic selection.

- c) Build and run the application to make sure it works. To test the probabilities, Change the values in `assignProbs()` to make one bitmap appear much more frequently than the others.

That's it - enjoy your slot machine game!

A Guide to Resources

Learning to use resources is an important part to Windows programming. Resources are binary files that are included with your executable when the project is built. Resources include bitmaps, cursors, icons, menus, string tables, and more. The basic purpose of resources is to provide a place to store such items in a simple manner. Using resources also prevents outside users from modifying your files. It is important to note that using resources does increase the size of your executable. If you are using a great number of bitmaps or other large multimedia files, you may want to consider storing them in a directory structure outside of the executable.

For this lab, we will focus on the use of bitmap resources. There are many ways to set up the resource files. In fact, the entire process may be done by hand. However, since we are using Visual C++, we will discuss its resource management tools. Visual C++ automatically handles resource management for you. Generally speaking, you won't need to modify any files by hand. The only exception is that you must `#include "resource.h"` into your main project file so that your application can see the generated resource files.

Adding a resource:

1. Once you have an appropriate resource file (your bitmap), you need to save it in an appropriate location (e.g. your project directory, or a subdirectory of your main project directory).
2. From this point, adding resources is straightforward. Go to View -> Resource View (CTRL + SHIFT + E) to bring up the Resource View.
3. Right-click on your project resources, select Add -> Add Resource from the menu.
4. You can select from the multiple blank resource files if you would like to create one of those types. However, we already have our resource file, so click on Import and browse to the location of your file.
5. Bring up the Properties window if it did not automatically appear (View -> Properties Window or F4).
6. In the Properties Window, you will find the attributes on your bitmap. It is suggested that you rename the ID field to something more descriptive than `IDB_BITMAP1`. However, to aid in identifying resources later on, it is a good practice to add the `IDB_` tag to the beginning of whatever name you chose. Also, resource names are generally all uppercase.
7. That's it! Now you have a resource that you can work with. Just make sure you include `resource.h` in your project file.

A Guide to using Bitmap Resources and the Windows Graphical Device Interface (GDI)

As you should know by now, a bitmap is basically a matrix of pixels, where each pixel stores color information. Our basic goal is to find a way to “paint” the bitmap pixels to our window. In order to paint the bitmap, we need to do a few things. First off, we need to have a handle to our bitmap. In this lab, you are given code that initializes the handles to the various bitmap resources (InitBitmapResources()). This code is quite simple:

==Code Fragment==

```
if ((handle_bitmap = (HBITMAP)LoadImage( hinstance,
                                         MAKEINTRESOURCE( IDB_BITMAP ),
                                         IMAGE_BITMAP,
                                         0, 0,
                                         LR_CREATEDIBSECTION)) == NULL)

    return( FALSE );
```

====End Code====

There is an IF statement around the load function to check to make sure the handle is valid after completion. LoadImage() returns a handle to an object so it must be casted as a HBITMAP to set up the bitmap handle properly. LoadImage takes an argument to the application (suggest that hinstance is a global handle to your application), the image resource ID (called the MAKEINTRESOURCE macro on your resource ID to get a proper integer ID), the flag IMAGE_BITMAP (to tell the function that you are passing a bitmap), two null values, and LR_CREATEDIBSECTION (to tell the function to use the bitmaps properties instead of forcing it to conform to the display).

Now that you have a handle to a bitmap, you need to display it. The provided function ShowBitmapResource() provides this functionality. Before you use the function, you must obtain a destination device context. A device context is a structure that stores information about graphical objects and your display device. We will concern ourselves with video display devices, but there are others such as printer displays. Before you enter the ShowBitmapResource() function, you need to obtain a device context for your current window. This is accomplished by the following (where hWnd is a global handle to your window):

==Code Fragment==

```
HDC destDC = GetDC(hWnd);
```

====End Code====

Now that you have a DC, you can pass that along with your bitmap handle and the coordinates of your paint location to the ShowBitmapResource() function.

The ShowBitmapResource() function is as follows:

==Code Fragment==

```
int ShowBitmapResource(HBITMAP hbitmap, HDC hDestDC, int xDest, int
yDest)
{
    HDC hSrcDC;           // source DC - memory device context
    BITMAP bmp;          // structure for bitmap info
    int nHeight, nWidth; // bitmap dimensions

    // create a DC for the bitmap to use
    if ((hSrcDC = CreateCompatibleDC(NULL)) == NULL)
        return(FALSE);

    // select the bitmap into the DC
    if (SelectObject(hSrcDC, hbitmap) == NULL)
        return(FALSE);

    // get image dimensions
    if (GetObject(hbitmap, sizeof(BITMAP), &bmp) == 0)
        return(FALSE);

    nWidth = bmp.bmWidth;
    nHeight = bmp.bmHeight;

    // copy image from one DC to the other
    if (BitBlt(hDestDC, xDest, yDest, nWidth, nHeight, hSrcDC, 0, 0,
        SRCCOPY) == NULL)
        return(FALSE);

    // kill the memory DC
    DeleteDC(hSrcDC);

    // return success!
    return(TRUE);
}
```

====End Code====

Most of the above code is explained by the comments. Essentially, you need to create a second DC that will store your bitmap information. Then you select the bitmap that you want to put in the DC and get the object information for the bitmap (dimensions in this case). Once that is completed, you simply copy the image from the bitmap DC to the destination DC (which paints it on the screen) and kill the old bitmap DC. If the function works, return true.

The last thing to mention about drawing the bitmaps is the re-drawing that will occur. If your window is taken out of focus (say you switch back to Visual Studio and your application goes into the background), then your bitmaps will no longer be visible. When this occurs, a new message is sent from the window to the message handler. It sends the message WM_PAINT which is a request to re-paint the window. The basic idea is to keep track of all the bitmaps that are visible and re-draw them all on a WM_PAINT.

For example,

==Code Fragment==

```
case WM_PAINT: //handle repaint case for when window loses focus
    PAINTSTRUCT ps; // declare a PAINTSTRUCT for use with this message
    HDC hdc;        // display device context for graphics calls
    hdc = BeginPaint(hWnd, &ps); // validate the window

    // your painting goes here!
    ShowBitmapResource(loc1, destDC, 30, 20);
    ShowBitmapResource(loc2, destDC, 140, 20);
    ShowBitmapResource(loc3, destDC, 250, 20);
    // stop painting
    EndPaint(hWnd, &ps); // release the DC
    break;
```

====End Code====

The WM_PAINT case declares some variables to work with, a PAINTSTRUCT and a HDC (display device context). Then it calls BeginPaint(). At this point, you simply repaint the bitmaps that you have been keeping track of. In this lab assignment, the loc1, loc2, and loc3 handles are handles to the currently displayed bitmaps, so it is only necessary to call ShowBitmapResource on each one of them.