

1 Windows Programming

1.1 Aims

You entered this course based on a pre-requisite of knowing Windows programming. In this module we will do a quick refresher and introduce COM for those that haven't seen it before.

1.2 Hungarian Notation



Pay particular attention to pages 55 – 58. This covers the naming conventions used by Microsoft which we will use as standard in this course. The code you write this semester should follow this notation scheme.

1.3 Basic Windows Programming



Have a flick through of the rest of Chapter 2.

If this is all new to you then you may have problems keeping up with this course. Understand this thoroughly before moving on the advanced material below.

1.4 Advanced Windows Programming



Read Chapter 3. The following sections are organised in the same order as the Chapter to give you some idea of emphasis.

1.4.1 Resources

These are handy for storing all the bits and pieces you need to make your game look good and sound good with having a gazillion data files lying around. Use them when you can. If you have massive amounts of data, or you want to make the data customisable, then find another way of organising it. For example, you could store all your resources in a separate binary data file which you can then load from manually.



Build the DEMO3_1 project for yourself. Have a look at the RC file from an ASCII editor. Using the IDE, apply some nonsense edit to one of the resources (e.g. add some pixels to the cursor) and then save it. Check out the ASCII text of the new RC file. Now try to compile it. Not having much

luck? Create your own resource file and header from the IDE, so that it is easier to maintain.



Build the DEMO3_2 project. Add a new sound resource, whatever you like. Modify the program so that it plays your new sound whenever a key is struck. This is why we need to get to DirectX in a hurry ...

1.4.2 Menus

Menus from resource files are just *too* easy. Make sure you know what's going on in Demos 3_3 and 3_4. Consider the lesson of Demo3_1 with respect to generating menus from the IDE. If you are using Visual Studio, I strongly recommend working with IDE from first instance to save a lot of grief later.

1.4.3 GDI

Even though we will do most of our graphics programming with DirectX, you will still find life a lot easier if you use GDI calls selectively. GDI can be excellent for quick overlay graphics to give you debugging output, particularly text. It's much slower than DirectX, but it's a quick way to get stuff up on the screen. Make sure you understand Demo3_5 as we will be using it for an exercise shortly.

1.4.4 Event Handling

It's clearly important to make your program behave sensibly when windows are moved, resized, closed etc. This also applies to windowed DirectX programs.



Fix up DEMO3_5 to draw the text up to the window boundaries, accounting for window resizing. Use the example of DEMO3_9 to help you out. While you're at it, add an annoying "Are you sure?" message box when the window closes.

1.4.5 Input Events

Clearly we need to be able to get stuff from the keyboard and mouse. DirectInput will offer some news ways of acquiring input that will fit better with the general game structure that we talked about in Module 1. DirectInput follows a similar model to `GetAsyncKeyState()` which is shown in Demo3_12. Here's a little exercise to get you working with the event driven way of dealing with input.



Using DEMO3_11 and DEMO3_13 as a base, write a little app to draw the character being pressed at the location indicated by the mouse. If the mouse is not in the current window, then don't draw anything. The app should be resizable.

1.4.6 Advanced GDI

Keep reading ... Chapter 4 now please.

In Chapter 4, the text provides some more detail of painting with pens and brushes. An important point when you are working with pens and brushes is that you must



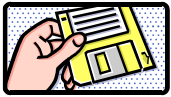
delete them if you are no longer going to use them. Otherwise you will chew up all your GDI resources for a bunch of pens and brushes you are not using. It is also good manners to destroy any global pens and brushes when closing your program.



Modify DEMO4_4 to operate with a randomly coloured background and a randomly coloured ellipse. The ellipse and the background should change to new random colours every time the ellipse bounces off the window edge.

1.4.7 Timing

Timer events are handy to know about, particularly if you want something to happen at regular intervals during your game. An issue that you will more often be concerned with is the animation rate of your program, which can be solved using `GetTickCount()` as shown in Demo4_7.



Make your modified DEMO4_4 run at 50 fps. Do it in a cleverer way than the text does so that you don't chew up all the CPU time. Run Task Manager to check.

1.4.8 Controls

While controls are definitely useful things to have if you want to set up dialog boxes etc. we don't really have time to visit them now. Note that you **definitely** want to use the resource editor when using controls, as the pain of modifying the resource files by hand quickly gets unbearable.

1.4.9 System Info

As outlined in the text, there is a heap of system info that you can pickup using `GetSystemMetrics()`. Particularly handy for working out window geometries.

1.5 Component Object Model

This is the key technology to making DirectX work. Once you understand COM, the rest of DirectX is pretty straightforward. The point of COM objects is that they are largely language independent, and can be upgraded without the need to re-compile code. You can also use COM objects to provide hardware independence, so that your code will run on lots of different devices without going a bunch of slow system calls. This is the key to DirectX. Sure, device drivers can provide a degree of hardware independence, but the system call model is slooow, as illustrated with GDI. By having COM objects tucking straight into the hardware, everything runs a lot faster; as we'll see in DirectX.

1.5.1 How It Works

Ultimately a COM library is a lump of binary code that can be run from any language. In order for you to use that code you need a way of working out which bit of binary code does what. COM has several key tricks for making this happen: Virtual Function Tables, interfaces and Globally Unique Identifiers.

The Virtual Function Tables are the key to pointing the right bit of code to the job. Virtual Function Tables are literally a structure of pointers to functions. Now, a function is a reference to a bit of executable binary, just as a variable is a reference to a bit of binary data. So, it makes sense that you can have pointers to functions just as you can have pointers to variables. For a pointer to a variable, the pointer contains the physical address in memory of the binary data. So, for a pointer to a function, the pointer contains the physical address of the executable binary code. A Virtual Function Table is therefore a list of the physical addresses of executable binary code. This explains how you work out where all the bits of code reside in memory.

COM presents this table of functions to you in an interface (if you know Java you should be familiar with the concept). The interface has many features that make it cool, but the ones we are most interested in are the function table, and the quirky ability to hook you into more interfaces.

But how do you find your interface in the swirling mass of COM? This is where the Globally Unique Identifiers come in. The interfaces to the bits of code are stored with GUIDs. The GUID is also provided in the header file for the COM interfaced software you wish to use (e.g. DirectX). When you query a COM object for an interface, it uses the GUID to match your query to the appropriate pointer and *voila* the COM delivers you a pointer to the table of functions you desire. How can a COM writer be sure that their GUID is truly globally unique? Microsoft has provided a utility called GUIDGEN.EXE that generates unique ids for everybody. Not too hard to do with 3.4×10^{38} ids to choose from.

1.5.2 How You Use It

The Microsoft approved way of using COM is to first call `CoInitialize()` to initialize COM, and then create your interface using `CoCreateInstance(...)`. In practice, many packages (including DirectX) tuck that away with a couple of macros (e.g. `DirectDrawCreate(...)`). In any case you will end up with an interface, which is really a pointer to a structure of functions and some data about the interface.

To actually call the functions provided by the interface (which is the reason you went to all the trouble of creating it), you have to remember that the interface is a pointer to a structure. To access some data in a structure to which you have a pointer, you use the notation:

```
i = dog->fleas;
```

which means that you have a pointer called `dog` which points to a structure with an element called `fleas`. To call a function in a structure to which you have a pointer, you use:

```
dog->scratch();
```

This is how you would call the `scratch` function from the interface `dog`.

Sitting underneath all COM is the `IUnknown` interface, which lets your program get pointers to other interfaces on a given object through the `QueryInterface()` method, and manage the existence of the object through the `AddRef()` and `Release()` methods. All other COM interfaces are inherited from `IUnknown`. These three methods in `IUnknown` are the first entries in the Virtual Function Table for every interface. (Sorry, I went all C++ here, but it really can't be helped.)

`QueryInterface()` tells you whether a COM interface is available; and if it is available gives you the pointer to the interface. COM needs to know whether or not its bits and pieces are being used. `AddRef()` tells COM that an interface is being used, so it had better be in memory. If there is more one app using an interface, `AddRef` keeps count. `Release()` undoes that count, and if the count goes to 0 lets COM clear the object from memory. You may need to know these details later on in DirectX, but these details are not used too often.

Because `QueryInterface()` sits in all the interfaces you can use it to access other interfaces. This is apparent when you look at the parameters to the method.

```
HRESULT QueryInterface(
    REFIID iid,           //Identifier of the requested interface
    void **ppvObject     //Address of output variable that receives the
                        //interface pointer requested in iid
);
```

(from the Platform SDK, COM and ActiveX Object Services)

The first parameter is the GUID that the COM object implementer generated when making his or her COM interface. The second parameter is the pointer to the structure that constitutes the interface.



Have a read of Chapter 5. The text is a little convoluted in presenting this material, but then again its pretty convoluting stuff. The DirectX references are good material for the next module.



Modify DEMO5_1 to include another interface called ME. This interface is to provide two functions: one called `winner()` that prints “I am the master of COM!” and another called `gloat()` that prints “Too easy ...”. Provide test code for your new functions in `main()`.

There is a lot, lot more to know about COM and the cool stuff it does. Having mastered this material however, you know enough to go and do some DirectX.

1.6 Preparation for Next Week



Read Chapters 6 and 7. Yes, all of it. Pages 239 – 396.