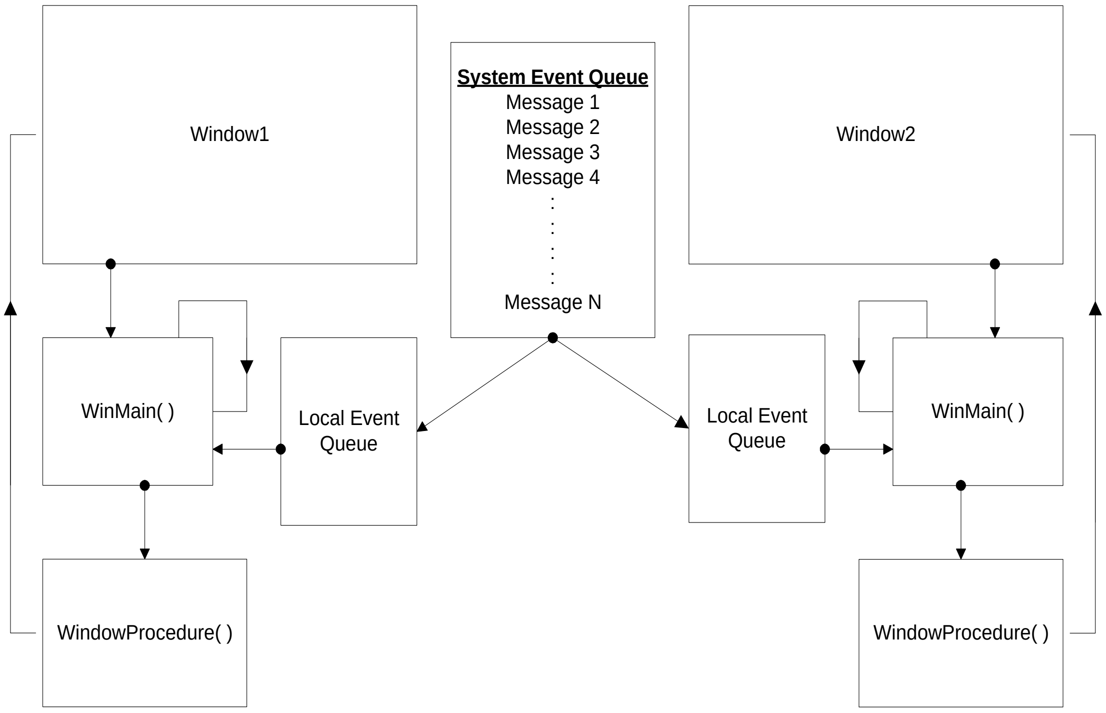
CS 529

Fundamentals of Game Development

1. Windows programming

1.1. Introduction

- Windows is a multitasking and multi-threaded OS
- Programs can be composed of a number of simpler threads of execution
- Windows is also an event driven OS
  - Windows programs wait for the user to take an action, then it fires an event
  - Events are sent to the OS
  - Some events are processed by the OS, the rest (which is the bigger part) are passed to running applications



1.2. WinMain function

- The WinMain function is the *initial entry point* for a windows-based application
- Similar to the main() function in a DOS program
- The WinMain function performs various initialization tasks
- Creating a window and processing events is done in the WinMain function
- The WinMain function is defined as follows:

  int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hprevinstance, LPSTR lpcmdline, int ncmdshow);

  The WINAPI declarator forces the parameters to be passed from left to right, in order to make sure that the startup code passes the parameters correctly to the functionalities.
  If the function succeeds, it should return the exit value contained in the message's wParam parameter (explained later), otherwise it returns 0 if it exits before entering the message loop.

- WinMain function parameters:
  - HINSTANCE hinstance: It is an instance handle that Windows *uniquely* generates for the application. It is a pointer used to track the running application. Its type is a pointer (void *)
  - HINSTANCE hPrevInstance: This parameter is obsolete now. It was used previously to keep track of the application that launched the current one.
    It is no longer used and is *always* NULL
  - LPSTR lpcmdline: It is a pointer to a null-terminated string (char) specifying the command line for the application. It is similar to the standard C & C++ main(int argc, char *argv), but without an argc parameter which was used to indicate the number of command line parameters.
    Example:
    A Windows application called MyApplication is launched with the following parameters: "MyApplication.exe param1 param2 param3 param4"

lpcmdline = “param1 param2 param3 param4”

As you can see, lpcmdline contains all the passed parameters in 1 string

- int ncmdshow: Indicates how the application window will be opened. It can be neglected and passed later on to the “ShowWindow” function.
  Examples: SW_HIDE, SW_SHOW, SW_SHOWNORMAL, SW_MAXIMIZE and SW_MINIMIZE…
  The SW prefix stands for “Show Window”

1.3. Window class

- A window is always created based on a *window class*
- There are some predefined window classes for list boxes, dialog boxes, buttons...
- More than one window can be created based on a single *window class* (button windows)
- When a window is created, additional characteristics unique to that window are defined
- To define a *window class*, define a structure of type WNDCLASSEX which contains window class information

```
typedef struct _WNDCLASSEX
{
    UINT        cbSize;
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HANDLE      hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
    HICON       hIconSm;
```

}WNDCLASSEX;

- WNDCLASSEX members:
  - **UINT cbSize**: Size of the structure
  - **UINT style**: This field specifies the class style flags. It can be any combination of the class style. The combination of the identifiers is done using the C bitwise *OR* operator (|). The most common identifiers are CS_HREDRAW and CS_VREDRAW (that indicate that all the windows created based on this window class are to be completely repainted whenever the horizontal or vertical window size of the window changes)
    The CS prefix stand for *Class Style*
  - **WNDPROC lpfnWndProc**: This field is a function pointer, which should save the address of the WndProc (window procedure) event handler function. This function is notified by the OS whenever something happens (A button is pressed...). This member and the WndProc function are explained in more details later on
  - **int cbClsExtra**: This field specifies the number of extra bytes to allocate following the window class structure. The system initializes this field to *zero*
  - **int cbWndExtra**: This field specifies the number of extra bytes to allocate following the window instance. The system initializes this field to *zero*.
    These two fields (cbClsExtra and cbWndExtra) reserve some extra space in the class structure and the window structure that Windows maintains internally. A program can use this extra space for its own purpose
  - **HANDLE hInstance**: Used to save a copy of the window handler
  - **HICON hIcon**: A handle to the class icon which will be used by the application
  - **HCURSOR hCursor**: A handle the cursor which will be displayed inside the window of the application. Use the "LoadCursor" function to retrieve a handle to a cursor from a resource or a predefined system cursor.
    HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName)
    HINSTANCE hInstance: Application handle. Set it to NULL to allow the default system cursors to be used
    LPCTSTR lpCursorName: Name string or a cursor resource identifier

- **HBRUSH hbrBackground**: This field holds a handle to the class background brush. It specifies the background color of the client area of the windows created based on this class
- **LPCTSTR lpszMenuName**: This field is a pointer to a null-terminated character string (char *) that specifies the resource name of the class menu. If this field is NULL, windows belonging to this class have no default menu
- **LPCTSTR lpszClassName**: This field is a pointer to a null-terminated string (char *) (the window class name). It is used to base the created window on the created class
- HICON hIconSm: This field holds a handle to a small icon that is associated with the window class. If this field is NULL, the system searches the icon resource for an icon of the appropriate size to use as the small icon

Implementation example:

```
WNDCLASSEX  wndclass;

wndclass.cbSize            = sizeof(WNDCLASSEX);
wndclass.style             = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc       = WndProc;
wndclass.cbClsExtra        = 0;
wndclass.cbWndExtra        = 0;
wndclass.hInstance     = hInstance;
wndclass.hIcon             = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground     = (HBRUSH)GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName      = NULL;
wndclass.hIconSm           = LoadIcon(NULL, IDI_APPLICATION);
wndclass.lpszClassName     = “CS529”;
```

1.2. Registering the Window class

- After creating a window class, the OS (Windows) should be informed about it
- This is done by registering the new window class using the “RegisterClassEx” function, which needs the address of the window class
- Example:

```
WNDCLASSEX wndclassex;
//Initialize wndclassex
...
...
RegisterClassEx(&wndclassex);
```

1.3. Creating the window

- The “CreateWindow” function creates a window or any object based on the same class
- HWND CreateWindow (LPCTSTR lpClassName, LPCTSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam);
    - LPCTSTR lpClassName: This field contains a null-terminated string (char *). It is the window class name that is registered with the RegisterClassEx function or any of the predefined system class names.
    - LPCTSTR lpWindowName: This field contains the window name that is displayed in the title bar. When creating a control, this field specifies the text of the control. When creating a control with an icon this field specifies the icon name or identifier
    - DWORD dwStyle: This field specifies the style of the window being created. It can be a combination of the window styles
    - int x and int y: These fields specify the initial position of the upper left corner of the window relative to the upper left corner of the screen. For a child window, the initial position of the upper left corner of the window relative to the upper left corner of the parent window’s client area
    - int nWidth: This field specifies the width of the window
    - int nHeight: This field specifies the height of the window

- HWND hWndParent: This field contains a handle to parent or owner window of the window being created. When a parent-child relationship exists between two windows the child window always appears on the surface of its parent
- HMENU hMenu: This field contains a handle to a menu
- HINSTANCE hInstance: This field contains the handle to the instance passed to the program as a parameter of WinMain function
- LPVOID lpParam: This field contains a pointer to a value to be passed to the window through the CREATESTRUCT structure.
- Implementation example:

```
hwnd = CreateWindow (    “CS529”,                  // window class name
 "Win32 Program",        // window caption
 WS_OVERLAPPEDWINDOW,    // window style
 CW_USEDEFAULT,          // initial x position
 CW_USEDEFAULT,          // initial y position
 CW_USEDEFAULT,          // initial x size
 CW_USEDEFAULT,          // initial y size
 NULL,                            // parent window handle
 NULL,                            // window menu handle
 hInstance,              // program instance handle
 NULL);                  // creation parameters
```

The CreateWindow function returns NULL if the window creation failed. On the other hand, if the window creation succeeds, a handle to the created window is returned

1.4. Displaying the window

- After creating the window using the “CreateWindow” function, it is not visible (unless we added the WS_VISIBLE flag). Whatever the case is, calling the “ShowWindow” function displays the window
- BOOL ShowWindow(HWND hwnd, int nCmdShow)
    - HWND hwnd: A handle to the window we want to show

  - int nCmdShow: This is the last variable that was passed by the WinMain function
- After showing the window, we should inform the OS to update the client area of our window by generating a WM_PAINT message. This is done by calling the “UpdateWindow” function
- BOOL UpdateWindow(HWND hwnd)
  - HWND hwnd: A handle to the window we want to update

1.5. Handling events

- The event handler is a callback function called by the OS (Windows) whenever an event occurs
- This callback function takes care of the events that are needed for it specific window, and passes the rest back to the OS
- When creating the window class, one of the members was **“WNDPROC lpfnWndProc”**, which is a pointer to the window's event handler: The WndProc function, which stands for Window Procedure
- Events are generated whenever the user or the OS perform tasks
- All events are saved in a common queue, and each window will have a private queue to save its own messages
- The main event loop (explained later) retrieves theses events (from the private queue) and send them to the WndProc function in order to be processed
- LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
  - HWND hwnd: Handle to the window receiving the message (event)
  - UINT iMsg: A 32-bit unsigned integer that represents the ID of the message being sent
  - WPARAM wParam & LPARAM lParam: They provide more information about the message. The use of these parameter depends on the message being sent.
- Messages example:
  - WM_CREATE: Sent when the application window is created. It can be used by the user in order to do some initializations or any other actions that should be done once upon starting the application

- WM_DESTROY: Sent when the application window is ready to be destroyed. This is usually occurs when the users manually closes the window. It can be used by the user to release all the loaded resources.
  The window procedure function responds to this message by calling the "PostQuitMessage" function which sends a WM_QUIT message that closes the application
  Recap: WM_DESTROY is sent when the window is closed, but keeps the application running! In order to close the application when its window is closed, call the "PostQuitMessage" function
- WM_PAINT: Sent when the window's content needs to be repainted. This usually occurs when when the window's position change or get resized, or when another window covers a part of it

▪ Window procedure implementation:

- RESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LLPARAM lParam)

```
{
 //Used when the WM_PATIN message is received
 PAINTSTRUCT ps;
 //A handle to a device context
 HDC hdc;

 switch(iMsg)
 {
 case WM_CREATE:
 //Initialize, load resources...
 return 0;//Success: Message has been handled
 break;

 case WM_PAINT:
 hdc = BeginPaint(hwnd, &ps);
```

```
                //Do all painting and rendering here
                EndPaint(hwnd, &ps);
                return 0; // Success: Message has been handled
                break;

            case WM_DESTROY:
                //Killing the application
                PostQuitMessage(0);
                return 0; // Success: Message has been handled
                break

            default:
                break;
        }
        //Return unprocessed messages to the OS
        return (DefWindowProc(hwnd, iMsg, wParam, lParam));
    }
```

- Returning a 0 is necessary to inform the OS that the current message has been handled

1.6. Main event (message) loop

- As mentioned previously, events are generated by the OS and saved in a general event queue
- Then each window will save its own private event queue
- These events are processed by the window procedure function (WndProc)
- But who fetches the messages from the private queue and sends it to the window procedure function?
- This is done in the main event loop, which is written within the "WinMain" function (Remember that the "WinMain" function is the entry point to our window application)

```
MSG msg;
while(GetMessage(&msg, NULL, 0, 0))
```

```
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
   }
```

- "GetMessage" fetches the next message from the application's private queue
  - Note that the message parameter here (msg) is different from the message sent to the window procedure function
- Here's the MSG structure:

```
typedef struct tagMSG
{
        HWND hwnd;
        UINT message;
        WPARAM wParam;
        LPARAM lParam;
        DWORD time;
        POINT pt;
} MSG;
```

  - HWND hwnd: Handle to the window where the message occurred
  - UINT message: Message ID (This is send to the WndProc function)
  - WPARM wParam: Sub qualifies message (Extra info) (This is send to the WndProc function)
  - LPARAM lParam: Sub qualifies message (Extra info) (This is send to the WndProc function)
  - DWORD time: Time when the event was fired
  - POINT pt: Mouse position
- Notice that the information received by the "WndPorc" function about a certain even is contained in this structure
- When a message is fetched from the queue using the "GetMessage" function, its time to call "TranslateMessage" which translates virtual-key messages into character messages

TranslateMessage(&msg);

- Finally, “DispatchMessage” is used to issue a call for the WndProc function passing along the current msg, which will eventually be processed

1.7. Switching to real-time: PeekMessage VS GetMessage

- In the previous example, we used “GetMessage” in order to fetch messages from the queue and process them
- “GetMessage” blocks the application. What that means is that the execution won't get inside the while loop until it receives a message. This is good for applications that don't need to be updated unless an event is fired
- Unfortunately, in real time applications, which virtually include all games, this is useless because the game keeps updating/rendering itself even if there no event like user input
- To solve this problem, “PeekMessage” should be used instead of “GetMessage”
  - “PeekMessage” checks for new messages in the queue. If there is, it processes them, otherwise, checking for messages stops and leaves time for other game components to be updated and rendered
  - In other words, the execution gets stuck inside “GetMessage” until an event occurs before proceeding, while “PeekMessage” is ended and exited even if there are no messages in the queue
- BOOL PeekMessage(LPMSG lpMsg, HWND hwnd, UINT wMsgFilterMin, UINT wMsgFilterMax, UINT wRemoveMsg);
  - LPMSG lpMsg: Pointer to a message structure
  - HWND hwnd: Handle to the window
  - UINT wMsgFilterMin: First message
  - UINT wMsgFilterMax: Last message
  - UINT wRemoveMsg: Removal message
  - Returns “0” if no more messages are available

- The last parameter parameter defines how “PeekMessage” handles the messages from the queue

- PM_NOREMOVE: Just checks for messages, but doesn't retrieve them from the queue. If there is a message, call “GetMessage” manually
- PM_REMOVE: Use it to allow “PeekMessage” to Check and retrieve messages. This is a perfect solution for real-time applications

```
MSG msg;
while(true)
{
 if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
 {
 if(msg.message == WM_QUIT)
 break;
                TranslateMessage(&msg);
                DispatchMessage(&msg); //Send the message to the WndProc function
      }

      Game_Update();
      Game_Render():
}
```

1.2. Implementation

```
#include "windows.h"
//WinMain function
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR
lpCmdLine, int nCmdShow)
{
 HWND hwnd;
 MSG msg;
```

```
    //Define a window class (a structure of type WNDCLASSEX)
    //Register the window class
    //Create a window (hWnd)
    //Displaying the created window (show and update Window)
    //Message Loop
    return (int)msg.wParam;
}

//Window procedure function
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
    switch (iMsg)
    {
        case WM_CREATE:
            return 0;

        case WM_COMMAND:
            return 0;

        case WM_PAINT:
            [process WM_PAINT message]
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```