

```

/*****
/*
/*----- M U L T I P L E . C -----*/
/* Task      : Demonstration of ITC mechanisms (Inter-Thread-Commu-
/*              nication). The model followed here is based on:
/*              1.) A thread that creates a window and dispatches
/*                  window messages. (While(GetMessage()))
/*              2.) Another thread that is created when the window is
/*                  created (WM_CREATE) and continually draws a graphic
/*                  (the tree of Pythagoras) in the window.
/*              Among other things, the difficulty consists of the fact
/*                  that when the window is closed, the thread created by
/*                  the window must also be terminated. This is achieved
/*                  by allocation of a kernel object of the Event type,
/*                  which is permanently polled by "PainterThread". Once
/*                  this event is signalled, the PainterThread must
/*                  terminate.
/*-----*/
/* Authors      : Michael Tischer and Bruno Jennrich
/* developed on   : 08/10/95
/* last update    : 08/31/95
/*****
#include <windows.h>
#include <math.h>

#include "turtle.h"

/* Window Extra Bytes -----*/
#define GWL_HPAINTERTHREAD 0
#define GWL_HTERMINATEEVENT 0 + sizeof( HANDLE )
#define GWL_MAX 2 * sizeof( HANDLE )

/* Parameter block that is passed to PainterThread: -----*/
typedef struct tagPYTHOPARMS
{
    HWND      hWnd;          /* Window in which the PainterThread is to paint */
    float      fAlpha;        /* Angle of "pythagorean" triangle */
    COLORREF   crColor;       /* color to be used */
    long       lSleep;        /* Inactive state after completion of a cycle */
    HANDLE      hTerminateEvent; /* Kernel event for thread termination */
    HANDLE      hAccessEvent;  /* Kernel event for safe structure access */
} PYTHOPARMS;
typedef PYTHOPARMS *PPYTHOPARMS;

/* Parameter block passed to the window by CreateWindow.
/* Used by threadCreateWindowFunction.
typedef struct tagCWFSTATUS
{
    LPSTR      lpCaption;      /* Caption of window */
    HANDLE      hWnd;          /* This item passes the handle of the window
/*                               to the creator of the window thread
    HANDLE      hevWndValid;   /* Event that signals the validity of the
/*                               handle placed in hWnd
    PYTHOPARMS PP;            /* Parameters for PainterThread created by
} CWFSTATUS;
/* the window
typedef CWFSTATUS *PCWFSTATUS;

/*****
/* Pythagoras: Draw subtree of tree of Pythagoras
/*-----*/
/* Parameter:      lEdge      : Edge length of subtree to be drawn
/*                  pPP        : Parameters (angle, etc. )
/*                  bDraw      : Draw subtree or calculate only coordinates
/*                               for BoundingRect (see Turtle32)
/* Return value: none
/*****
void Pythagoras( LONG Edge, PPYTHOPARMS pPP, BOOL bDraw )
{
    if( Edge < 4 ) return;          /* Subtree too small, so end recursion */

```

```

if( WAIT_OBJECT_0 ==                                /* Terminate thread? */
    WaitForSingleObject( pPP->hTerminateEvent, 0L ) )
    return;

turtleForward( ( float )Edge, bDraw );                /* lower square edge */
turtleRotate( ( float )90.0 );

turtleForward( ( float )Edge, bDraw );                /* right edge */
turtleRotate( ( float )90.0 );

turtleForward( ( float )Edge, bDraw );                /* upper edge */

turtlePush();          /* the triangle is added here, so save Turtle */

turtleRotate( ( float )90.0 );
turtleForward( ( float )Edge, bDraw );                /* left edge */

turtlePop();          /* back to end of upper edge */

/* Set orientation of turtle according to angle of triangle -----*/
turtleRotate( -( 180 - pPP->fAlpha ) );

/* save current status, because now the left subtree -----*/
/* is going to be drawn. -----*/
turtlePush();
Pythagoras( ( long )(cos( ( pPP->fAlpha * PI ) / 180.0 ) * Edge),
            pPP, bDraw );          /* draw left subtree */
turtlePop();

turtleForward( ( float )(cos( ( pPP->fAlpha * PI ) / 180.0 ) * Edge),
            FALSE );          /* go to apex of right angle */
turtleRotate( ( float )-90.0 );

Pythagoras( ( long )(sin( ( pPP->fAlpha * PI ) / 180.0 ) * Edge),
            pPP, bDraw );          /* draw right subtree */
}

/*****
/* threadPainterFunction: Workhorse of PainterThread */
/*-----*/
/* Parameter:    lpStart : Address of a PYTHOPARMS structure through */
/*                which the painter gains access to the */
/*                required parameters. */
/* Return value: Thread Exit Code */
/*-----*/
/* Info: This thread is created in WM_CREATE of the window. If the */
/*        window is closed, an event also signals the request to */
/*        exit the thread. */
*****/
DWORD WINAPI threadPainterFunction( LPVOID lpStart )
{
    /* The creator of the thread waits until the thread signals */
    /* that the parameters passed in lpStart are no longer being */
    /* referenced. The creator passes a pointer to local variables */
    /* in lpStart. If the creator thread continued running, then the */
    /* local variables would lose their validity. If this thread then */
    /* accesses local variables of the creator that no longer exist, */
    /* an access violation occurs! */
    /* */

    /* create local copy of passed parameters -----*/
    PYTHOPARMS PP = *(( PPYTHOPARMS )lpStart);

    /* Signal end of access to parameters to creator -----*/
    SetEvent( PP.hAccessEvent );

    /* Lower priority of this thread a little, so that creator can */
    /* manage user input at "full speed". */
    SetThreadPriority( GetCurrentThread(), THREAD_PRIORITY_BELOW_NORMAL );

    /* Initialize turtle of this thread -----*/
    turtleSetWindow( PP.hWnd );

    /* Set color or pen -----*/
    turtleSetPen( PP.crColor, 1 );

```

```

/* First run a "blind" pass to determine the          */
/* drawing area.                                     */
turtleInitBounding();

turtleMoveTo( 0, 0, FALSE );                          /* Set starting position */
turtleSetAngle( ( float )0.0 );
/* draw the tree once and determine the bounding rectangle */
Pythagoras( 100, &PP, FALSE);

/* Terminate the thread here already? */
if( WAIT_OBJECT_0 ==
    WaitForSingleObject( PP.hTerminateEvent, 0 ) )
    return 0;

/* The following graphic output has the same dimensions as the */
/* ones determined in the blind passage. Therefore, map these */
/* dimensions to the entire Client Area of the window.          */
turtleUseBounding( TRUE );

while( TRUE )                                          /* The never ending story? */
{
    InvalidateRect( PP.hWnd , NULL, TRUE );          /* Clear background */
    UpdateWindow( PP.hWnd );

    turtleMoveTo( 0, 0, FALSE );                      /* Draw new tree */
    turtleSetAngle( ( float )0.0 );
    Pythagoras( 100, &PP, TRUE );

    /* Terminate thread because the creator is signalling */
    /* the appropriate event, or simply wait for inactive */
    /* state (lSleep)? */
    if( WAIT_OBJECT_0 ==
        WaitForSingleObject( PP.hTerminateEvent, PP.lSleep ) )
        return 0;
}
}

/*****
/* CreatePainterThread: Creates the PainterThread */
/*-----*/
/* Parameter:    pPP : Address of a PYTHOPARMS structure containing */
/*               the parameters of the tree. */
/* Return value: Thread ID */
/*****
HANDLE CreatePainterThread( PPYTHOPARMS pPP )
{
    DWORD dwThreadID;

    pPP->hAccessEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
    if( pPP->hAccessEvent )
    {
        HANDLE hThread = CreateThread( NULL,
                                        0L,
                                        threadPainterFunction,
                                        pPP,
                                        0L,
                                        &dwThreadID );

        /* Wait until the PainterThread has created a copy of the */
        /* variables passed by pointer. */
        WaitForSingleObject( pPP->hAccessEvent, INFINITE );
        CloseHandle( pPP->hAccessEvent );
        return hThread;
    }
    return NULL;
}

/*****
/* WndProc: Window procedure */
/*-----*/
/* Parameter:    Default window procedure parameters */
/* Return value: Message dependent */
/*-----*/
/* Info: With the exception of WM_CREATE, WM_DESTROY and WM_PAINT, */
/*         all messages are passed on to the default window procedure */
/*         (DefWndProc). WM_PAINT is only called to demarcate the */

```

```

/*      areas to be repainted (UpdateRect), to remove a WM_PAINT      */
/*      message from the message queue.                                */
/*****
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wP, LPARAM lP )
{
    switch( uMsg )
    {
        /* When the window is created, PainterThread is also created      */
        case WM_CREATE:
        {
            /* A PYTHOPARMS structure is passed in the call -----*/
            LPCREATESTRUCT lpCS = ( LPCREATESTRUCT ) lP;
            PPYTHOPARMS pPP = (PPYTHOPARMS )lpCS->lpCreateParams;

            /* In the structure of Pythoparms, the handle of an event is      */
            PainterThread that tells the Painter when          */
            /* it must finish its work.                                     */
            pPP->hTerminateEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
                                   /* manual, non signalled event */

            /* In addition, the window in which the Painter is -----*/
            /* to paint gets passed. -----*/
            pPP->hWnd = hWnd;

            /* Able to create the event ? -----*/
            if( pPP->hTerminateEvent )
            {
                HANDLE hThread;
                /* Note handle of Terminate Event in extra bytes of      */
                /* window for later access (see WM_DESTROY).            */
                /* In addition, the PainterThread is created and its Thread */
                /* ID is noted in the extra bytes.                        */
                SetWindowLong( hWnd,
                               GWL_HTERMINATEEVENT,
                               (LONG ) pPP->hTerminateEvent );
                hThread = CreatePainterThread( pPP );
                SetWindowLong( hWnd,
                               GWL_HPAINTERTHREAD,
                               ( LONG )hThread );

                /* If able to create thread, then everything's OK -----*/
                if( !hThread )
                {
                    /* Otherwise, delete Terminate event again -----*/
                    /* and output message. -----*/
                    CloseHandle( pPP->hTerminateEvent );
                    MessageBox( hWnd,
                                "Cannot create Painter Thread",
                                "Error",
                                0 );
                }
                return 0;
            }
            else MessageBox( hWnd,
                            "Cannot create Terminate event",
                            "Error", 0 );

            return -1;
        }
        break;

        case WM_DESTROY:
        {
            /* Able to create a PainterThread ? -----*/
            if( GetWindowLong( hWnd, GWL_HPAINTERTHREAD ) )
            {
                HANDLE hTerminate, hThread;
                /* Signal request for termination to Painter      */
                hTerminate=(HANDLE)GetWindowLong(hWnd,GWL_HTERMINATEEVENT);
                SetEvent( hTerminate );

                /* If thread terminates, its HANDLE is signalled -----*/
                hThread = ( HANDLE )GetWindowLong( hWnd, GWL_HPAINTERTHREAD );
                WaitForSingleObject( hThread, INFINITE );

                CloseHandle( hTerminate );
                CloseHandle( hThread );
            }
        }
    }
}

```

```

        PostQuitMessage( 0 );                                /* Exit MessageLoop */
    }
    break;
case WM_PAINT:
{
    PAINTSTRUCT ps;                                          /* Remove UpdateRectangles from list */
    BeginPaint( hWnd, &ps );
    EndPaint( hWnd, &ps );
}
break;
}
return DefWindowProc( hWnd, uMsg, wP, lP );
}

/*****
/* threadCreateWindowFunction: MessageLoop Thread */
/*-----*/
/* Parameter:    lpCWFStatus : Parameters for creating the window */
/*               and tree (caption, angle etc. ) */
/* Return value: Thread Exit Code */
/*-----*/
/* Info: This thread creates a window and edits the message */
/*        queue until the window is closed. */
/*        The window must be closed "from the outside" through a */
/*        call to DestroyWindow. The window handle required for this */
/*        is returned in the hWnd item of the lpStatus structure. */
/*        This function signals the availability of the window handle */
/*        rough the hevWndValid item */
/*****
DWORD WINAPI threadCreateWindowFunction( PCWFSTATUS lpCWFStatus )
{
    MSG msg;
    HWND hWnd;

    /* Create window of the "MultiPle" class -----*/
    hWnd = CreateWindow( "MultiPle",
                        lpCWFStatus->lpCaption,
                        WS_OVERLAPPED | WS_THICKFRAME,
                        0,0,300,300,
                        HWND_DESKTOP,
                        NULL,
                        NULL,
                        &lpCWFStatus->PP );    /* Pass tree parameters */

    lpCWFStatus->hWnd = hWnd;    /* Transfer window handle to lpStatus */
    SetEvent( lpCWFStatus->hevWndValid );    /* and signal availabil- */
                                           /* ity to caller */

    if( hWnd )    /* Able to create window ? */
    {
        ShowWindow( hWnd, SW_NORMAL );    /* Display window */
        UpdateWindow( hWnd );    /* execute immediate repaint */
        BringWindowToTop( hWnd );    /* Window to foreground */

        while( GetMessage( &msg, NULL, 0,0 ) )    /* Message loop */
        {    /* is ended by PostQuitMessage() */
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        return TRUE;
    }
    return FALSE;
}

/*****
/* CreateWindowThread: Creates a thread that creates a window in */
/* which another thread draws in the tree of */
/* Pythagoras. */
/*-----*/
/* Parameter:    lpCaption : Window title */
/*               crColor    : Color the tree will be drawn in */
/*               fAlpha      : Angle of tree */
/*               lsleep      : Inactive state after draw cycle (in ms) */
/* Return value: Window handle of the window created in the thread */
/*-----*/

```

```

/* Info: This function is called "outside", that is, from a process. The return of the window handle makes it the responsibility of the calling process to close the window. If the process does not close a window created here, it will continue doing its job until the system is shut down. The calling process closes the window by calling the CloseWindowThread.
*/
/*****
HWND WINAPI CreateWindowThread( LPSTR lpCaption,
                                COLORREF crColor,
                                float fAlpha,
                                long lSleep )
{
    DWORD dwThreadID;
    HANDLE hThread;
    CWFSTATUS cwfStatus;

    /* Initialize status structure -----*/
    cwfStatus.hevWndValid = CreateEvent( NULL, TRUE, FALSE, NULL );
    cwfStatus.lpCaption = lpCaption;

    cwfStatus.PP.crColor = crColor;
    cwfStatus.PP.fAlpha = fAlpha;
    cwfStatus.PP.lSleep = lSleep;

    if( cwfStatus.hevWndValid ) /* Able to create event ? */
    {
        hThread = CreateThread( NULL, 0L, threadCreateWindowFunction,
                                &cwfStatus, 0L, &dwThreadID );

        /* Wait until WindowHandle is valid -----*/
        WaitForSingleObject( cwfStatus.hevWndValid, INFINITE );
        CloseHandle( cwfStatus.hevWndValid ); /* Close event */
        return cwfStatus.hWnd;
    }
    return NULL;
}
/*****
/* CloseWindowThread: Closes window and all the threads associated with it.
*/
/*-----*/
/* Parameter: hWnd : Handle of window to be closed
/* Return value: none
*/
/*-----*/
/*****
void WINAPI CloseWindowThread( HWND hWnd )
{
    SendMessage( hWnd, WM_DESTROY, 0, 0 );
}

/*****
/* DllMain: Main entry point of DLL (replaces LibMain)
*/
/*-----*/
/* Parameter: none
/* Return value: TRUE - Able to pop context
/* FALSE - Stack underflow
*/
/*****
BOOL WINAPI DllMain(HANDLE hModule, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
        {
            /* When a new process logs in, the window class must be registered. Under Win32 there is no longer a PrevInstance, so you don't have to take existing window classes into consideration.
            WNDCLASS wc;

            /* Complete redraw on resizing -----*/
            wc.style = CS_HREDRAW | CS_VREDRAW;
            wc.lpfnWndProc = WndProc;
            wc.cbClsExtra = 0;
            /* Number of extra bytes in window structure (Thread handle and Terminate event).
            wc.cbWndExtra = GWL_MAX;

```

```

    wc.hInstance      = NULL;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor( NULL, IDI_APPLICATION );
    wc.hbrBackground  = ( HBRUSH )(COLOR_BTNFACE + 1);
    wc.lpszMenuName    = NULL;
    wc.lpszClassName  = "MultiPle";
    return RegisterClass( &wc );
}
break;

case DLL_THREAD_ATTACH:
break;

case DLL_THREAD_DETACH:
break;

case DLL_PROCESS_DETACH:
    /* Unregister window class. -----*/
    UnregisterClass( "MultiPle", NULL );
    break;
}
return TRUE;
}

```