

```

/*****
/*
/*-----
/* Task      : Implementation of a LOGO turtle. Each thread that uses
/*             this DLL has its own turtle. The state information of
/*             each thread's turtle is stored via thread local storage
/*             slots (TLS).
/*-----
/* Authors    : Michael Tischer and Bruno Jennrich
/* developed on : 08/10/95
/* last update  : 09/01/95
*****/

#include <windows.h>
#include <math.h>
#include <assert.h>

#include "turtle.h"

/* Indexes of thread local storage slots. With a different approach,
/* a structure could be allocated by means of VirtualAlloc(), with
/* the address of this memory block stored in only one TLS slot.
/* However, this is contradicted by the fact that VirtualAlloc()
/* allocates at least 4K. Therefore, at processor level a more
/* efficient memory management (similar to malloc()) would have
/* to be used.
DWORD tlsidxX;           /* current X coordinate */
DWORD tlsidxY;           /* current Y coordinate */
DWORD tlsidxAngle;       /* current angle */
DWORD tlsidxLineWidth;   /* line width */
DWORD tlsidxColor;       /* line color */
DWORD tlsidxPen;         /* current pen */
DWORD tlsidxWnd;         /* current output window */
DWORD tlsidxStackMem;    /* stack memory */
DWORD tlsidxStackPtr;    /* stack pointer */
DWORD tlsidxBoundingLeft; /* Bounding rectangle, left border */
DWORD tlsidxBoundingTop; /* Bounding rectangle, top border */
DWORD tlsidxBoundingRight; /* Bounding rectangle, right border */
DWORD tlsidxBoundingBottom; /* Bounding rectangle, bottom border */
DWORD tlsidxUseBounding; /* Adapt output to bounding rectangle */

/*== Macros for simplifying TLS access =====*/
#define GetX() ((int)TlsGetValue(tlsidxX))
#define GetY() ((int)TlsGetValue(tlsidxY))
#define GetAngle() ((float)(LONG)TlsGetValue(tlsidxAngle))
#define GetLineWidth() ((int)TlsGetValue(tlsidxLineWidth))
#define GetColor() ((COLORREF)TlsGetValue(tlsidxColor))
#define GetPen() ((HPEN)TlsGetValue(tlsidxPen))
#define GetWnd() ((HWND)TlsGetValue(tlsidxWnd))
#define GetStackMem() ((PTURTLECONTEXT)TlsGetValue(tlsidxStackMem))
#define GetStackPtr() ((LONG)TlsGetValue(tlsidxStackPtr))
#define GetBoundingLeft() ((LONG)TlsGetValue(tlsidxBoundingLeft))
#define GetBoundingTop() ((LONG)TlsGetValue(tlsidxBoundingTop))
#define GetBoundingRight() ((LONG)TlsGetValue(tlsidxBoundingRight))
#define GetBoundingBottom() ((LONG)TlsGetValue(tlsidxBoundingBottom))
#define GetUseBounding() ((LONG)TlsGetValue(tlsidxUseBounding))

#define SetX(v) assert(TlsSetValue(tlsidxX, (LPVOID)(v)))
#define SetY(v) assert(TlsSetValue(tlsidxY, (LPVOID)(v)))
#define SetAngle(v) assert(TlsSetValue(tlsidxAngle, (LPVOID)(LONG)(v)))
#define SetLineWidth(v) assert(TlsSetValue(tlsidxLineWidth, (LPVOID)(v)))
#define SetColor(v) assert(TlsSetValue(tlsidxColor, (LPVOID)(v)))
#define SetPen(v) assert(TlsSetValue(tlsidxPen, (LPVOID)(v)))
#define SetWnd(v) assert(TlsSetValue(tlsidxWnd, (LPVOID)(v)))
#define SetStackMem(v) assert(TlsSetValue(tlsidxStackMem, (LPVOID)(v)))
#define SetStackPtr(v) assert(TlsSetValue(tlsidxStackPtr, (LPVOID)(v)))
#define SetBoundingLeft(v) assert(TlsSetValue(tlsidxBoundingLeft, (LPVOID)(v)))
#define SetBoundingTop(v) assert(TlsSetValue(tlsidxBoundingTop, (LPVOID)(v)))
#define SetBoundingRight(v) assert(TlsSetValue(tlsidxBoundingRight, (LPVOID)(v)))
#define SetBoundingBottom(v) assert(TlsSetValue(tlsidxBoundingBottom, (LPVOID)(v)))
#define SetUseBounding(v) assert(TlsSetValue(tlsidxUseBounding, (LPVOID)(v)))

/*== Maximum number of context items on the stack =====*/
#define MAX_TURTLESTACK 100

/*****

```

```

/* turtleInit: Put turtle in original state. */
/*-----*/
/* Parameters: none */
/* Return value: none */
/*-----*/
/* Info: This function is automatically called when */
/* a new thread is started whose parent process */
/* links this DLL. Never call the function directly, */
/* because the stack memory is also allocated here. */
/******/
void WINAPI turtleInit( void )
{
    PTURTLECONTEXT pStack; /* Pointer for stack memory */

    SetX( 0 );
    SetY( 0 );
    SetAngle( ( float ) 0 );
    SetLineWidth( 0 );
    SetColor( RGB( 0,0,0 ) );
    SetPen( CreatePen( PS_SOLID, GetLineWidth(), GetColor() ) );
    SetWnd( 0 );
    SetUseBounding( FALSE );
    turtleInitBounding();

    pStack = VirtualAlloc( NULL, /* Allocate stack */
                          sizeof( TURTLECONTEXT ) * MAX_TURTLESTACK,
                          MEM_COMMIT,
                          PAGE_READWRITE );
    SetStackMem( pStack );
    SetStackPtr( 0 );
}

/******/
/* turtleInitBounding: Initializes the bounding rectangle */
/*-----*/
/* Parameters: none */
/* Return value: none */
/*-----*/
/* Info: This function initializes the coordinates of the bounding */
/* rectangle. The maximum and minimum drawing positions are */
/* entered in this rectangle. This way you can implement a */
/* "blind" passage which you can first enter the required */
/* drawing area in the bounding rectangle. You can use this */
/* rectangle for subsequent drawing operations to scale the */
/* display, so that the drawing always takes up the same */
/* amount of space in the client area of a */
/* window. */
/* (see DLLMain) */
/******/
void WINAPI turtleInitBounding( void )
{
    SetBoundingLeft( 32767 ); /* minimum and maximum */
    SetBoundingTop( 32767 ); /* permissible coordinates */
    SetBoundingRight( -32768 );
    SetBoundingBottom( -32768 );
    SetUseBounding( FALSE );
}

/******/
/* turtleUseBounding: Sets flag for using BoundingRect */
/*-----*/
/* Parameter: bUse : True - Scale output to BoundingRect */
/* False - Output to absolute MM_TEXT */
/* coordinates */
/* Return value: none */
/*-----*/
/* Info: Bounding is used in the following way: */
/* turtleInitBounding(); */
/* invisible Paint (turtleForward()); */
/* turtleUseBounding( TRUE ); */
/* Paint... */
/******/
void WINAPI turtleUseBounding( BOOL bUse )
{
    SetUseBounding( bUse );
}

```

```

/*****
/* turtleExit: Release memory and GDI objects .
/*-----
/* Parameters: none
/* Return value: none
/*-----
/* Info: This function is automaticall called whenever a thread
/* terminates. (see DLLMain)
/*****
void WINAPI turtleExit( void )
{
    if( GetStackMem() ) /* Does stack memory exist? */
        VirtualFree( GetStackMem(),
                      sizeof( TURTLECONTEXT ) * MAX_TURTLESTACK,
                      MEM_DECOMMIT );
    /* Does pen exist ? */
    if( GetPen() ) DeleteObject( GetPen() );
}

/*****
/* turtleSetPen: Creates new current pen.
/*-----
/* Parameters: crCol : Color of new pen
/* lLineWidth: Width of pen
/* Return value: none
/*****
void WINAPI turtleSetPen( COLORREF crCol, LONG lLineWidth )
{
    /* release any existing pen */
    if( GetPen() ) DeleteObject( GetPen() );
    /* Place pen in TLS */
    SetPen( CreatePen( PS_SOLID, lLineWidth, crCol ) );
    SetColor( crCol ); /* Place color and width in TLS */
    SetLineWidth( lLineWidth );
}

/*****
/* turtleSetWindow: Set output window of thread.
/*-----
/* Parameters: hWnd : Handle of output window
/* Return value: none
/*-----
/* Info : Each turtle thread can manage only one output window.
/* To display output in more than one window, you have to
/* specify the output window using turtleSetWnd(). All the
/* drawing commands that follow will affect this
/* window.
/* Extension : Before a GDI command is executed in turtleForward(),
/* the DC is initialized with the required objects.
/* Each time, the pen to be used is selected and
/* the scaling mode (BoundingRect) is set. However,
/* this is quite inefficient. On the other hand,
/* by using a window with the style CS_OWNDC, the DC
/* only needs to be updated when the pen or the drawing
/* area is changed. Windows saves the state of the DCs
/* for such windows internally, so that GetDC()
/* returns the calls of a
/* preinitialized DC.
/*****
void WINAPI turtleSetWindow( HWND hWnd )
{
    SetWnd( hWnd );
}

/*****
/* turtleSaveContext: Place state of turtle in structure.
/*-----
/* Parameters: pTC - Address of a TURTLECONTEXT structure that
/* receives current turtle state.
/* Return value: none
/*-----
/* Info : The LOGO turtle works, to a great extent, independently
/* of coordinates. To return to a special state of the
/* turtle, an opportunity to save the current state must be
/* created. After saving the current state, you can continue

```

```

/*      drawing in order to return to the "starting point" later      */
/*      (see RestoreContext). The BoundingRect is not saved,          */
/*      instead, it is a thread-global                                */
/*      "Resource".                                                    */
/*****
void WINAPI turtleSaveContext( PTURTLECONTEXT pTC )
{
    assert( pTC );
    pTC->lX      = GetX();
    pTC->lY      = GetY();
    pTC->fAngle   = GetAngle();
    pTC->lLineWidth = GetLineWidth();
    pTC->crColor  = GetColor();
    pTC->hPen     = GetPen();
    pTC->hWnd     = GetWnd();
}

/*****
/* turtleRestoreContext: Restore old state of turtle                  */
/*-----*/
/* Parameters:    pTC - Address of a TURTLECONTEXT structure that was */
/*                previously initialized by turtleSaveContext.         */
/* Return value: none                                                  */
/*****
void WINAPI turtleRestoreContext( PTURTLECONTEXT pTC )
{
    assert( pTC );
    SetX( pTC->lX );
    SetY( pTC->lY );
    SetAngle( pTC->fAngle );
    SetLineWidth( pTC->lLineWidth );
    SetColor( pTC->crColor );
    SetPen( pTC->hPen );
    SetWnd( pTC->hWnd );
}

/*****
/* turtleRotate: Change turtle's orientation or direction of movement */
/*-----*/
/* Parameter:    fAngle - Angle by which the turtle is to be rotated, */
/*                in degrees. Positive values rotate counter-         */
/*                clockwise                                             */
/* Return value: none                                                  */
/*****
void WINAPI turtleRotate( float fAngle )
{
    /* Limit angle to values between 0 and 360 */
    SetAngle( ( float )fmod( GetAngle() + (360.0 - fAngle) , 360.0 ) );
}

/*****
/* turtleSetAngle: Set absolute orientation                            */
/*-----*/
/* Parameter:    fAngle - Angle at which turtle is to "look",         */
/*                in degrees.                                           */
/* Return value: none                                                  */
/*****
void WINAPI turtleSetAngle( float fAngle )
{
    SetAngle( ( float )fmod( fAngle + 360.0, 360.0 ) );
}

/*****
/* turtleForward: Move turtle                                         */
/*-----*/
/* Parameters:    fLineLen - Length of distance to be covered         */
/*                bDraw    - should a line be drawn during            */
/*                movement?                                            */
/* Return value: none                                                  */
/*****
void WINAPI turtleForward( float fLineLen, BOOL bDraw )
{
    double dRadiant;
    POINT pStart;
    HDC hDC = GetWnd() ? GetDC( GetWnd() ) : 0;

```

```

assert( hDC );

/* degrees to radiant */
dRadiant = ( GetAngle() / 180.0 ) * 3.141592654;
/* If a line needs to be drawn, the starting point of the line ---*/
/* must be saved. ---*/
if( ( bDraw ) && ( hDC ) )
{
    pStart.x = GetX();
    pStart.y = GetY();
}

/* Calculate new position based on the current angle -----*/
SetX( GetX() + ( int )( cos( dRadiant ) * fLineLen ) );
SetY( GetY() + ( int )( sin( dRadiant ) * fLineLen ) );

/* Update BoundingRect -----*/
if( GetX() < GetBoundingLeft() ) SetBoundingLeft( GetX() );
if( GetX() > GetBoundingRight() ) SetBoundingRight( GetX() );
if( GetY() < GetBoundingTop() ) SetBoundingTop( GetY() );
if( GetY() > GetBoundingBottom() ) SetBoundingBottom( GetY() );

if( ( bDraw ) && ( hDC ) ) /* Draw line? */
{
    /* Can be refined (see turtleSetWnd() and CS_OWNDC ) -----*/
    HPEN hOldPen = SelectObject( hDC, GetPen() );
    int iMap;

    if( GetUseBounding() ) /* Use BoundingRect? */
    {
        RECT r;

        /* Map all coordinates of BoundingRect to ClientArea of -----*/
        /* window. -----*/
        GetClientRect( GetWnd(), &r );
        /* Distort scaling -----*/
        iMap = SetMapMode( hDC, MM_ANISOTROPIC );

        /* logical coordinates = BoundingRect -----*/
        SetWindowOrgEx( hDC, GetBoundingLeft(), GetBoundingTop(), NULL );
        SetWindowExtEx( hDC, ( GetBoundingRight() - GetBoundingLeft() ),
                        ( GetBoundingBottom() - GetBoundingTop() ),
                        NULL );

        /* Map to ClientArea -----*/
        SetViewportOrgEx( hDC, 0, 0, NULL );
        SetViewportExtEx( hDC, (r.right - r.left),
                        (r.bottom - r.top),
                        NULL );
    }

    MoveToEx( hDC, pStart.x, pStart.y, NULL ); /* Draw line */
    LineTo( hDC, GetX(), GetY() );

    SelectObject( hDC, hOldPen ); /* Select old pen */

    if( GetUseBounding() ) /* Restore old drawing mode */
        SetMapMode( hDC, iMap );
}
if( hDC ) ReleaseDC( GetWnd(), hDC );
}

/*****
/* turtleMoveTo: Set current drawing position */
/*-----*/
/* Parameters:  lX    - new X coordinate */
/*              lY    - new Y coordinate */
/*              bDraw - draw a line at the new */
/*                  point? */
/* Return value: none */
/*****/
void WINAPI turtleMoveTo( LONG lX, LONG lY, BOOL bDraw )
{
    POINT pStart;
    HDC hDC = GetWnd() ? GetDC( GetWnd() ) : 0;

    assert( hDC );

```

```

/* If a line needs to be drawn, the starting point of the line ---*/
/* must be saved. ---*/
if( ( bDraw ) && ( hDC ) )
{
    pStart.x = GetX();
    pStart.y = GetY();
}

SetX( lX ); /* Set new position */
SetY( lY );

/* Update BoundingRect -----*/
if( GetX() < GetBoundingLeft() ) SetBoundingLeft( GetX() );
if( GetX() > GetBoundingRight() ) SetBoundingRight( GetX() );
if( GetY() < GetBoundingTop() ) SetBoundingTop( GetY() );
if( GetY() > GetBoundingBottom() ) SetBoundingBottom( GetY() );

if( ( bDraw ) && ( hDC ) ) /* Draw line? */
{
    /* Can be refined. (see turtleSetWnd() and CS_OWNDC ) -----*/
    HPEN hOldPen = SelectObject( hDC, GetPen() );
    int iMap;

    if( GetUseBounding() ) /* Use BoundingRect? */
    {
        RECT r;

        /* Map all coordinates of BoundingRect to ClientArea of -----*/
        /* window. -----*/
        GetClientRect( GetWnd(), &r );
        /* Distort scaling -----*/
        iMap = SetMapMode( hDC, MM_ANISOTROPIC );

        /* logical coordinates = BoundingRect -----*/
        SetWindowOrgEx( hDC, GetBoundingLeft(), GetBoundingTop(), NULL );
        SetWindowExtEx( hDC, ( GetBoundingRight() - GetBoundingLeft() ),
                        ( GetBoundingBottom() - GetBoundingTop() ),
                        NULL );

        /* Map to ClientArea -----*/
        SetViewportOrgEx( hDC, 0, 0, NULL );
        SetViewportExtEx( hDC, (r.right - r.left),
                        (r.bottom - r.top),
                        NULL );
    }

    MoveToEx( hDC, pStart.x, pStart.y, NULL ); /* Draw line */
    LineTo( hDC, GetX(), GetY() );

    SelectObject( hDC, hOldPen ); /* Select old pen */

    if( GetUseBounding() ) /* Restore old drawing mode */
        SetMapMode( hDC, iMap );
}
if( hDC ) ReleaseDC( GetWnd(), hDC );
}

/*****
/* turtlePush: Push current turtle state onto custom TurtleStack */
/*-----*/
/* Parameters: none */
/* Return value: TRUE - Able to push context */
/* FALSE - Stack overflow */
*****/
BOOL WINAPI turtlePush( void )
{
    assert( GetStackPtr() < MAX_TURTLESTACK );

    if( GetStackPtr() < MAX_TURTLESTACK )
    {
        PTURTLECONTEXT pTC;
        pTC = GetStackMem(); /* Get stack */
        turtleSaveContext( &pTC[ GetStackPtr() ] ); /* Save context */
        SetStackPtr( GetStackPtr() + 1 ); /* Increment stack pointer */
        return TRUE;
    }
}

```

```

    return FALSE;
}

/*****
/* turtlePop: Pop current turtle state from stack */
/*-----*/
/* Parameters:    none */
/* Return value: TRUE  - Able to pop context */
/*               FALSE - Stack underflow */
*****/
BOOL WINAPI turtlePop( void )
{
    assert( GetStackPtr() > 0 );

    if( GetStackPtr() > 0 )
    {
        PTURTLECONTEXT pTC;
        pTC = GetStackMem(); /* Get stack */
        SetStackPtr( GetStackPtr() - 1 ); /* Decrement stack pointer */
        /* Restore context -----*/
        turtleRestoreContext( &pTC[ GetStackPtr() ] );
        return TRUE;
    }
    return FALSE;
}

/*****
/* DllMain: Main entry point of DLL (replaces LibMain) */
/*-----*/
/* Parameters:    none */
/* Return value: TRUE  - Able to pop context */
/*               FALSE - Stack underflow */
*****/
BOOL WINAPI DllMain(HANDLE hModule, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason) /* Why was DllMain called? */
    {
        /* A newer process wants to use the services of the DLL. */
        /* Thus the process must allocate the necessary TLS indexes */
        /* for all threads running in its charge. */
        /* This can vary from process to process, but since */
        /* each process has its own DLL global variables, */
        /* the threads can operate different processes */
        /* with different TLS indexes. */
        case DLL_PROCESS_ATTACH:
            /* DLL_PROCESS_ATTACH is the only reason whose return value */
            /* is considered, therefore, on incorrect TLS slot allocation, */
            /* FALSE must be returned. */
            if( !(tlsidxX = TlsAlloc()) ||
                !(tlsidxY = TlsAlloc()) ||
                !(tlsidxAngle = TlsAlloc()) ||
                !(tlsidxLineWidth = TlsAlloc()) ||
                !(tlsidxColor = TlsAlloc()) ||
                !(tlsidxPen = TlsAlloc()) ||
                !(tlsidxWnd = TlsAlloc()) ||
                !(tlsidxStackPtr = TlsAlloc()) ||
                !(tlsidxStackMem = TlsAlloc()) ||
                !(tlsidxBoundingLeft = TlsAlloc()) ||
                !(tlsidxBoundingTop = TlsAlloc()) ||
                !(tlsidxBoundingRight = TlsAlloc()) ||
                !(tlsidxBoundingBottom = TlsAlloc()) ||
                !(tlsidxUseBounding = TlsAlloc()) )
            {
                return FALSE;
            }
            return TRUE;
        break;

        case DLL_THREAD_ATTACH:
            /* For each new thread the turtle must be initialized ----*/
            turtleInit();
        break;

        case DLL_THREAD_DETACH:
            /* After thread termination, turtle variables (Stack, GDI */
            /* objects) must be released. */
            turtleExit();
        break;
    }
}

```

```

case DLL_PROCESS_DETACH:
    /* When the process "dies", the TLS slots aren't required any */
    /* longer either. */
    TlsFree( tlsidxX );
    TlsFree( tlsidxY );
    TlsFree( tlsidxAngle );
    TlsFree( tlsidxLineWidth );
    TlsFree( tlsidxColor );
    TlsFree( tlsidxPen );
    TlsFree( tlsidxWnd );
    TlsFree( tlsidxStackMem );
    TlsFree( tlsidxStackPtr );
    TlsFree( tlsidxBoundingLeft );
    TlsFree( tlsidxBoundingTop );
    TlsFree( tlsidxBoundingRight );
    TlsFree( tlsidxBoundingBottom );
    TlsFree( tlsidxUseBounding );
    break;
}
return FALSE;
}

```