

Wiggly Worms

By Shridhar Kumar (shriprem@gmail.com)

Program Description:

Wiggly Worms is a PowerBuilder sample application that compares the costs and benefits of using multiple threads versus a single thread. With multiple core processor chips getting main-stream, I became interested in exploring the multithreaded features offered in PowerBuilder.

The application is named after the worm like shapes that wiggle around on the canvas inside a container window.

The multithreads have been implemented in this app using only 2 layers: the container window and the thread objects themselves with no brokering inbetween. The brokering is not needed since the thread objects alter the container window's canvas using device context handle.

GUI Features:

- The **Lives** slider will let you control how many worms will float around in the window. *Range: 1 thru' 10.*
- The **Length** slider will let you control the length (in pixels) of the worms. *Range 25 thru' 250.*
- The **Length Upto** checkbox lets you specify whether all worm lengths are *equal to* or *upto* the Length slider setting.
- The **Thickness** slider will let you specify the thickness (in pixels) of the worms. *Range 1 thru' 11 in steps of 2.*
- The **Thickness Upto** checkbox lets you specify whether all worm thicknesses are *equal to* or *upto* the Thickness slider setting.
- The **MultiThreaded** checkbox will let you control whether all worms are implemented in a single thread or separate threads.
- The **Change Color** button will let you change the worms' random colors.
- The **Thread Count** text field will display the count of threads owned by the application process. Clicking on this text field will display additional process information in a message box.
- **Resizing the window** will also resize the canvas.
- All controls on the window are also accessible via **keyboard accelerators**

Implementation:

- The core functionality of the application is implemented using only two WinAPI calls: **GetDC()** for getting the canvas's device context and **SetPixel()** for drawing pixels that make up the worm.
- The application also uses the native PowerBuilder functions: **SharedObjectRegister()**, **SharedObjectGet()** and **SharedObjectUnregister()** calls for multithreading.
- The Thread Count functionality has been implemented with only a 3 line code using WMI calls.

Observations:

I have tested this application on a desktop PC with dual Pentium 3.4 GHz / 1GB RAM, a desktop PC with single Pentium 2.8 GHz / 512 MB RAM and a laptop with single Pentium 2.8 GHz / 512 MB RAM.

1. While in single thread all the worms move in sync. In multithread mode, on the dual Pentium PC, the worms moved in sync when upto 3 lives. With more lives or on the single CPU machines, the worms were terribly out of sync.
2. The worms moved relatively faster in multithreaded mode, although maybe out of sync.
3. In multithreaded mode, the worms continue moving when the container window is being dragged around or when the process/thread information message box is displayed.
4. Keyboard accelerators were more responsive than using mouse to click on the controls in multithreaded mode.
5. The application crashed once in a while, randomly, when multithread modes were used.
6. The ThreadCount increases every time the number of lives is changed or when switching between the single/multithread modes. During my tests, I have let the thread counts reach upto 200 before exiting the app and verifying in the SysInternal's Process Explorer (replacement for Windows Task Manager) that all the threads were indeed closed.
7. I found that using a SetNull() on the thread object was critical, in addition to the more obvious DESTROY and SharedObjectUnregister() calls, to cause the application to clean up all the spawned threads when it exited.
8. I also found that using the **SuspendThread()** WinAPI call instead of SetNull() also had the same effect. Although, using both SetNull() and **SuspendThread()** calls together did not make any difference in decreasing thread count (see next note).
9. I think internal references to the thread handles are being kept alive in the PBVM and this is why the closed threads are not going away until the

application fully exits. I tried using the **TerminateThread()** WinAPI call with no effect.

10. I have also included a window **w_wiggly_worms_local**, which is a version of the **w_wiggly_worms** but the references to thread objects are handled via local variables instead of the instance variables. I did this to see if referencing the thread objects via a local variable would cause the threads to disappear after they are closed. No improvement was noted.

Recommendations:

1. Sybase should put in some work to make sure that any references to the thread handles that are being maintained internally in the PBVM are closed out so that the OS can terminate the threads.

Sybase needs to overhaul PowerBuilder's multithreaded features or at least make the current implementation more robust. This should gain higher priority from Sybase in view of the multicore CPUs getting main-stream.

2. I also found that the **HtrackBar** does not have **Enabled** property setting. This is also the case with **HScrollBar**, **VScrollBar** and **VtrackBar**. I am surprised with this finding. Sybase should fix this and expose the Enabled property for all these 4 controls. (At the same time, I do agree with Sybase that the two ProgressBar controls do not need the Enabled property exposed.)