Playing with the Windows Animation Manager: Fixing a sample

devblogs.microsoft.com/oldnewthing/20130314-00

March 14, 2013

Raymond Chen

Windows 7 provides a component known as the Windows Animation Manager, known to some people by its acronym WAM, pronounced "wham". There are some nice sample programs for WAM on MSDN, but for some reason, the authors of the samples decided to animate the three color components of a resultant color.

Because apparently the authors of those sample programs can look at a color and say, "Oh, clearly the red component of this color increases gradually at first, then speeds up its rate of increase, and then slows back down until it reaches its final value; while simultaneously the blue component is doing the opposite, but over a shorter time span, and the green component is remaining fixed."

Today's exercise is to fix the sample program so you can actually *see* and *understand* what WAM is doing, rather than just watching psychedelic colors change and saying, "Gee, that's pretty."

But first, some background:

Windows Animation is a component which manipulates *variables*. A *variable* is a number which varies over time. You tell Windows Animation things like "I would like you to animate this variable from 1 to 10 over the next 7 seconds." You can then interrogate the variable for its current value, and it might say "Right now, the value is 6."

The idea is that each of these variables is connected to some visual property, like the position of an object. When you paint the object, you consult the current value of the variable to find out where you should draw it.

One of the annoying bits about Windows Animation is that you have to set up a bunch of stuff just to get things started. You need an *animation manager*, which is the object that runs the show. You also need an *animation timer* whose job is to tell the animation manager what

time it is. (Under normal circumstances, you would use the default timer, which records real-world time, but you might want to replace it with a special timer for debugging that runs at half-speed, or maybe one which varies its speed based on how fast you clap.)

Okay, back to fixing the sample.

Start with the Timer-Driven Animation and make these changes:

```
                // disable the initial animation
                // Fade in with Red
                // hr = ChangeColor(COLOR_MAX, COLOR_MIN, COLOR_MIN);


HRESULT CMainWindow::DrawBackground(
    Graphics &graphics,
    const RectF &rectPaint
    )
{
    // Get the RGB animation variable values


    INT32 red;
    HRESULT hr = m_pAnimationVariableRed->GetIntegerValue(
        &red
        );
    if (SUCCEEDED(hr))
    {
        INT32 green;
        hr = m_pAnimationVariableGreen->GetIntegerValue(
            &green
            );
        if (SUCCEEDED(hr))
        {
            INT32 blue;
            hr = m_pAnimationVariableBlue->GetIntegerValue(
                &blue
                );
            if (SUCCEEDED(hr))
            {
                // Replace the drawing code as follows
                SolidBrush brushBackground(Color(255, 255, 255));
                hr = HrFromStatus(graphics.FillRectangle(
                    &brushBackground,
                    rectPaint
                    ));


                SolidBrush brushCircle(Color(0, 0, 0));
                hr = HrFromStatus(graphics.FillEllipse(
                    &brushCircle,
                    red, green, 10, 10
                    ));
            }
        }
    }


    return hr;
}
```

Instead of drawing a psychedelic background color, I draw a small circle using the old `red` value as the x-coordinate, and the old `green` value as the y-coordinate. I didn't rename the variables or get rid of the unused `blue` variable because I wanted to make as few changes as possible.

Run this program, and click to make the circle move. Observe that when the circle moves, it starts slowly, then accelerates, and then decelerates as it gets closer to its final location. What's more, if you click while the circle is still moving, the circle demonstrates *inertia* as it turns to head toward its new target location.

I bet you never noticed the acceleration, deceleration, or inertia in the original background-color version.

With a little bit of work, you can make the sample even more interesting by making the circle go to *where you clicked*. It looks like a lot of work when I spell it out below, but most of it consists of *deleting* code.

First, do a search/replace and rename `m_pAnimationVariableRed` to `m_pAnimationVariableX`, and rename `m_pAnimationVariableGreen` to `m_pAnimationVariableY`. Delete `m_pAnimationVariableBlue` entirely, as well as any references to it. I decided to just bite the bullet and deal with the consequences of renaming/deleting variables.

Now we can simplify the `CMainWindow::CreateAnimationVariables` method so all it does is create the two coordinate variables.

```
HRESULT CMainWindow::CreateAnimationVariables()
{
    HRESULT hr = m_pAnimationManager->CreateAnimationVariable(
        0,
        &m_pAnimationVariableX
        );
    if (SUCCEEDED(hr))
    {
        hr = m_pAnimationManager->CreateAnimationVariable(
            0,
            &m_pAnimationVariableY
            );
    }


    return hr;
}
```

We want the circle to move when you click the mouse, so let's do that. Delete `CMain-Window::OnLButtonDown` and change the window procedure so that clicks move the circle.

```
LRESULT CALLBACK CMainWindow::WndProc(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
    )
{
    …
        case WM_LBUTTONDOWN:
            {
                pMainWindow->ChangePos(
                    (SHORT)LOWORD(lParam),
                    (SHORT)HIWORD(lParam)
                 );
            }
            return MESSAGE_PROCESSED;
    …
}
```

And rename the member function `ChangeColor` to `ChangePos`, and instead of taking red and green, have it take x and y.

```
HRESULT CMainWindow::ChangePos(
    INT x,
    INT y
    )
{
    const UI_ANIMATION_SECONDS DURATION = 0.5;
    const DOUBLE ACCELERATION_RATIO = 0.5;
    const DOUBLE DECELERATION_RATIO = 0.5;


    // Create a storyboard


    IUIAnimationStoryboard *pStoryboard = NULL;
    HRESULT hr = m_pAnimationManager->CreateStoryboard(
        &pStoryboard
        );
    if (SUCCEEDED(hr))
    {
        // Create transitions for the position animation variables


        IUIAnimationTransition *pTransitionX;
        hr = m_pTransitionLibrary->CreateAccelerateDecelerateTransition(
            DURATION,
            x,
            ACCELERATION_RATIO,
            DECELERATION_RATIO,
            &pTransitionX
            );
        if (SUCCEEDED(hr))
        {
            IUIAnimationTransition *pTransitionY;
            hr = m_pTransitionLibrary->CreateAccelerateDecelerateTransition(
                DURATION,
                y,
                ACCELERATION_RATIO,
                DECELERATION_RATIO,
                &pTransitionY
                );
            // delete former “blue” transition
            if (SUCCEEDED(hr))
            {
                // Add transitions to the storyboard


                hr = pStoryboard->AddTransition(
                    m_pAnimationVariableX,
                    pTransitionX
                    );
                if (SUCCEEDED(hr))
                {
                    hr = pStoryboard->AddTransition(
```

```
                        m_pAnimationVariableY,
                        pTransitionY
                        );
                    // delete former “blue” transition
                    if (SUCCEEDED(hr))
                    {
                        // Get the current time and schedule the storyboard for play
                        UI_ANIMATION_SECONDS secondsNow;
                        hr = m_pAnimationTimer->GetTime(
                            &secondsNow
                            );
                        if (SUCCEEDED(hr))
                        {
                            hr = pStoryboard->Schedule(
                                secondsNow
                                );
                        }
                    }
                }


                // delete former “blue” transition


                pTransitionY->Release();
            }


            pTransitionX->Release();
        }


        pStoryboard->Release();
    }


    return hr;
}
```

Now you can click the mouse on the client area, and the dot will chase it like a puppy.

The basic idea behind the Windows Animation Library is that for each property you want to animate, you associate an animation variable, and when you want to perform the animation, you create a transition for each variable describing how you want the animation to proceed, put all the transitions into a storyboard, and then schedule the storyboard.

Of course, you can build optimizations on top of the basic idea. For example, you might not create the animation variable until the first time you need to animate the property. Another optimization is invalidating only the parts of the window that need repainting, rather than

invalidating the entire client area. You can do this by registering a change handler on your variables: When the change handler notifies you that a value changed, invalidate the old position and the new position. This will erase the old location and draw at the new location.

Next time, I'll build a program that animates a hundred objects, just for fun.

Raymond Chen

Follow