Transparency in Windows Forms Controls

Let’s talk about controls on a Form. Each child control occupies some rectangular area with coordinates (Left, Top, Width, Height) on its parent control. Usually, the whole rectangle is filled with a solid color assigned to the BackColor property. The value of BackColor can be derived from the parent control or set explicitly. It works OK till the time when we want to display non-solid background for the parent control or we need to show some child controls partially overlapped. For example, if the Form has some background image and there is a CheckBox control on that Form the user may expect to see the part of the background image behind the CheckBox, not just solid background color. Is it possible with Windows Forms controls? Yes, sure. Just set the CheckBox.BackColor property to a Color with the Alpha channel less than 255, for example, to Color.Transparent (with Alpha = 0). Can we always be successful with this approach? Not exactly. Let’s look at the first sample (TransparencyTest1).



This Form contains a few child controls and the background image. There are CheckBox, C1Gauge, PictureBox, and TrackBar controls. The PictureBox shows the big round button that can be moved over the Form and clicked with the mouse. The BackColor property is set to Color.Transparent for all controls but the TrackBar. Some controls are overlapped.

What’s wrong with this sample? There are a few things:

1. If the rectangular bounds of two controls with transparent background overlap each other the intersection shows the top control and the background image. It doesn’t show the bottom control.
2. Some controls, such as TrackBar, TextBox, ListBox, and others, don’t support transparent or semi-transparent background colors.
3. While moving the PictureBox control over the Form you can observe how slowly it redraws and you can see visual artifacts – some parts of the background image are cleared then redrawn.
4. Hit-testing works incorrectly for overlapped controls. The top control in the z-order processes mouse messages even for the transparent areas. Those messages should be targeted at the controls subsequent in the z-order.
5. The background image cannot be animated. For example, it is possible to animate a .gif image with the PictureBox control. But loading the same .gif image into the Form.BackgroundImage shows only the first .gif frame as a static image.

Why it works in such a way? Because Windows Forms don’t support real transparency for the child controls. It tries to simulate it with a trick. For example, when the value of the CheckBox.BackColor has the Alpha channel less than 255, the control is drawn with the following algorithm:

* The internal graphics buffer is created and filled with the background color of the parent control, such as a Form.
* If the parent control has the background image the corresponding part of that image is drawn to the graphics buffer.
* The control itself is drawn to the graphics buffer with the specified BackColor.
* The graphics buffer is drawn to the parent control as an opaque image.

As we can see, the built-in support for non-rectangular and semi-transparent controls is very limited in Windows Forms. Is there any way to overcome this limitation?

There are a few options. For example, we could create a custom hierarchy of lightweight visual elements drawn on some specific control. This is an interesting way and we used it in the ComponentOne Ribbon and other complex controls. But it makes difficult to reuse the existing WinForms controls as part of the visual hierarchy. Also, it requires handmade design-time support.

As an alternative way, we can see using layered windows as child controls. Layered windows were introduced in Windows 2000 to improve performance and visual effects for a window that has a complex shape, animates, or wishes to use alpha-blending effects. Unfortunately, this feature was supported for top-level windows only, not for child windows. The situation has been changed with the release of Windows 8 which adds support for the WS\_EX\_LAYERED style to child windows. And even for the older versions of Windows, such as Windows 7, we can use the clipped popup layered windows to simulate transparent child controls on a Form.

Let’s look at the second sample (TransparencyTest2).



It consists of the same controls as the first sample. Each of the child controls meant to be transparent is now added to the special UserControl-based TransparentHost which is a layered window. Also, we don’t set the Form’s background image. Instead, an opaque PictureBox was added (docked to the Form) to animate the .gif image behind the other child controls.

The TransparentHost control doesn’t support transparency at the design-time. It is actually not different from the regular UserControl on the designer surface in Visual Studio. The magic begins when you call the TurnOnTransparency() method at runtime when the Form has already been initialized. That converts the TransparentHost control to either layered popup window (for Windows 7) or to layered child window (for Windows 10).

BTW: To properly use the TransparentHost control on Windows 8.1 and Windows 10 make sure you added the app.manifest to your application project (click “Project – Add New Item… – General – Application Manifest File” in the menu of Visual Studio). Among the other parameters, the manifest must explicitly set Windows 8.1/10 as supported operating systems:

<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">

 <application>

 <!-- Windows 8.1 -->

 <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}" />

 <!-- Windows 10 -->

 <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}" />

 </application>

</compatibility>

If you don’t include those settings to the manifest, the layered child windows will not be allowed even on Windows 10 and the layered popup windows will be used instead.

All the mentioned issues with the transparent controls are now gone (the second one requires special attention, see below). But there are a few new ones. The layered window painted with the UpdateLayeredWindow API is just a bitmap, it doesn’t invalidate or redraw itself each time when some child control is updated. We need to explicitly call the TransparentHost.Invalidate() method each time when something is changed in the visual appearance of the hosted child controls. That makes impractical to add, for example, a WebBrowser control to the TransparentHost because we don’t know for sure when it needs to be repainted. Working with complex controls, like grids, can be tricky since we need to track all their events following the visual updates. In the sample the TrackBar control changes appearance of its thumb element when the mouse pointer hovers it. There is no the corresponding event – we had to call the Invalidate() method each time when the mouse pointer moves over the TrackBar.

Another problem is how to achieve transparency for controls which cannot be drawn to a transparent bitmap. Some controls have no such a problem. For example, if we set C1Gauge.BackColor to Color.Transparent, then obtain the image with the GetImage() method, the resulting bitmap will be semitransparent. To draw C1Gauge efficiently we could override the TransparentHost.RenderControls method as follow:

protected override void RenderControls(Graphics g)

{

 using (Image img = c1Gauge1.GetImage())

 {

 g.DrawImage(img, c1Gauge1.Left, c1Gauge1.Top);

 }

}

But most of the WinForms controls don't have a method like C1Gauge.GetImage. The only way to obtain their image is sending the WM\_PRINT message to the window (the standard Control.DrawToBitmap method utilizes that message). It turns out that “printing” controls to the transparent background causes bad problem in many cases. For example, we cannot do it with the CheckBox control because it draws text with the ClearType technology which makes text distorted on the transparent bitmap.

To alleviate this problem the TransparentHost control “prints” the child controls with the background of TransparentHost (if other is not set explicitly on the child controls), then it replaces the background color with Color.Transparent in the resulting image. It works well if you set the TransparentHost.BackColor property to the color which doesn’t differ much from the color of the background surface behind the TransparentHost.

One more problem may occur when working with the layered popup windows on Windows 7, for example. Usually, controls become focused after clicking with the mouse. That activates the popup window and makes the owner Form deactivated. This is a limitation and has no workaround but you can activate the Form again with the TransparentHost. ActivateHostParent() method when necessary.

And the last but not the least note about TransparentHost. Never use the following properties of the TransparentHost control at runtime: Parent, Location, Size, Width, Height, Left, Top, Right, Bottom. Their values can be unpredictable. Instead, use the following properties: HostParent, HostLocation, HostSize. Also, use the BringHostToFront() method instead of BringToFront().

See the source code of the TransparencyTest2 sample for more details.