Surface Dial

Lab 3. Creating an On Screen Visual for the Surface Dial

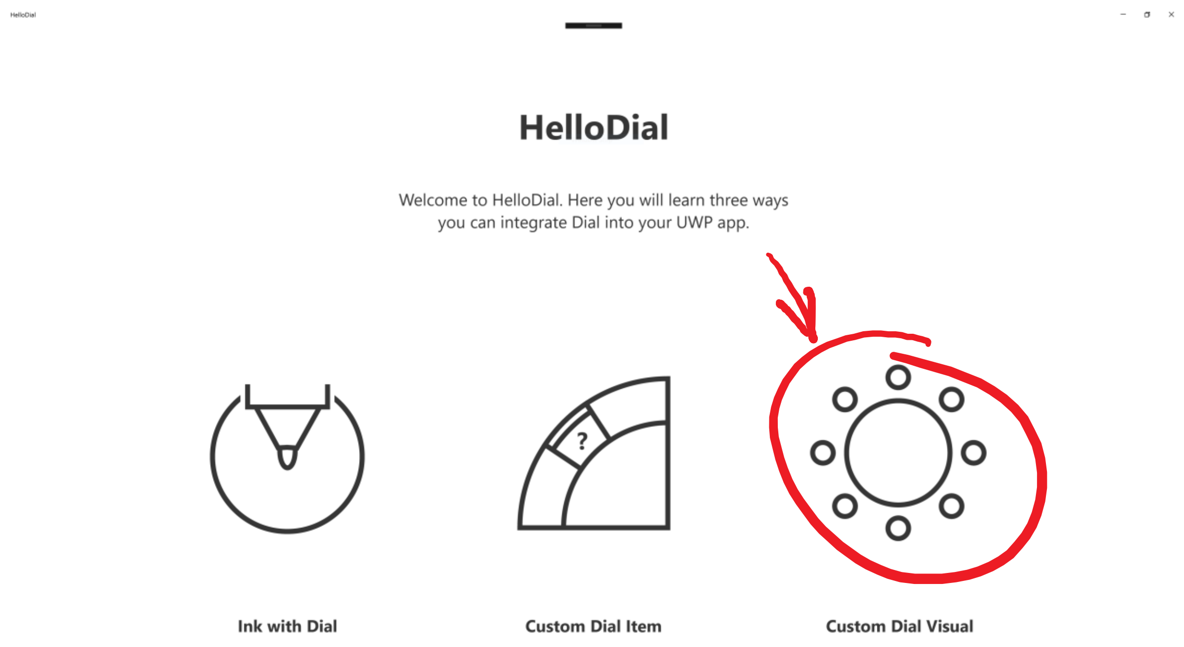
The Surface Dial can interact directly on the screen of a Surface Studio. The default behavior will display a **RadialControllerMenu** around the Surface Dial. Using the lessons learned in the second lab you can modify the **RadialControllerMenu** by adding your own items.

There are scenarios where you may wish to replace the **RadialControllerMenu** with your own control. It is recommended that you enable your own controls for an on-screen experience as a second level menu item. The first experience a customer will get with the Surface Dial on your application should be consistent with their other app experiences. The customer can then invoke a menu item to display your own control around the Surface Dial.

In this lab you will learn how to display a **UserControl** around the Surface Dial when it is placed on the screen of a Surface Studio.

This lab assumes you have a working knowledge of C# and the Visual Studio IDE. To test the code in this lab you will need a Surface Dial and a Surface Studio.

1. Open the existing application:
    a. Find the folder named **5. Start HelloDial Screen Visual**
    b. In this folder open the **HelloDial.sln** solution file in **Visual Studio 2015 or Visual Studio 2017**
2. Build and Run the application (F5) and you will see the start screen. In this hands-on lab you will build the third scenario 'Custom Dial Visual'


HelloDial
HelloDial
Welcome to HelloDial. Here you will learn three ways you can integrate Dial into your UWP app.
Ink with Dial
Custom Dial Item
Custom Dial Visual

3. In **Solution Explorer** open the **Pages** folder. You will see InkPage.xaml and MenuItemPage.xaml which were created in 'Lab 1: Dial with Ink' and 'Lab 2: Dial Menu Item'. Instead, open the **OnScreenVisualPage.xaml** file. You will see a Page that has already been created which contains a back **Button**.

4. Open the **OnScreenVisualPage.xaml.cs** code file. In this file you will see code that looks similar to the code you wrote in the previous lab. The code adds a new **RadialControllerMenuItem** and handles the menu item being invoked. Also the handler for the back **Button** is already implemented.

5. Return to the **OnScreenVisualPage.xaml** file and add a Canvas control. This Canvas will host the UserControl when the Surface Dial is placed on the screen.

```
<Grid Background="White">
       <Button Background="Transparent"  Click="BackButton_Click"
            FontFamily="Segoe MDL2 Assets" Content="&#xE72B;" VerticalAlignment="Top"
            HorizontalAlignment="Left" Margin="5"/>
       <TextBlock FontFamily="Segoe UI" FontSize="66.5" FontWeight="Bold"
            Foreground="#353535"
            Margin="0,150,0,0" TextAlignment="Center" Text="Custom Dial Visual" />
       <Canvas x:Name="DialCanvas" />
   </Grid>
```

6. Open the **OnScreenVisualPage.xaml.cs** file and add a member variable to maintain a reference to a **DialColorControl**.

```
public sealed partial class OnScreenVisualPage : Page
    {
        private RadialController myController;
        private RadialControllerMenuItem screenColorMenuItem;
        private DialColorControl dialControl;
       . . .
```

7. To the namespace at the top of the class add:

```
using HelloDial.Controls;
```

8. In the **Controls** folder of the solution you can find the code for the **DialColorControl**. The **DialColorControl** is a **UserControl** that displays a ring of gradient color. The ring can be rotated to select a different color.

```
public DialColorControl()
        {
            this.InitializeComponent();
            //handedness for on-screen UI
            Windows.UI.ViewManagement.UISettings settings = new
Windows.UI.ViewManagement.UISettings();
```

```
            rightHanded = (settings.HandPreference ==
Windows.UI.ViewManagement.HandPreference.RightHanded);
            //if left handed then rotate the entire control 180
            if (!rightHanded)
            {
                RotateTransform trans = new RotateTransform();
                trans.Angle = 180;
                colorControl.RenderTransform = trans;
            }
        }
```

9. Return to the **Pages** folder of the solution.

10. Go to the **OnScreenVisualPage** class. At the end of the constructor add code to handle the **ScreenContact** events and the **RotationChanged** event. These methods will be added in the next steps. The **ScreenContact** events will be raised when the Surface Dial is placed on the screen, moved, and removed from the screen.

```
public OnScreenVisualPage()
{
    this.InitializeComponent();

    RadialControllerConfiguration myConfiguration =
        RadialControllerConfiguration.GetForCurrentView();
    myConfiguration.SetDefaultMenuItems(new[]
        {
            RadialControllerSystemMenuItemKind.Volume,
            RadialControllerSystemMenuItemKind.Scroll
        });

    // Create a reference to the RadialController.
    myController = RadialController.CreateForCurrentView();

    // Create an icon for the custom tool.
    RandomAccessStreamReference icon =
      RandomAccessStreamReference.CreateFromUri(
      new Uri("ms-appx:///Assets/dial_icon_custom_visual.png"));

    // Create a menu item for the custom tool.
    screenColorMenuItem =
      RadialControllerMenuItem.CreateFromIcon("Screen Color", icon);

    // Add the custom tool to the RadialController menu.
    myController.Menu.Items.Add(screenColorMenuItem);

    screenColorMenuItem.Invoked += ColorMenuItem_Invoked;

    myController.ScreenContactStarted += MyController_ScreenContactStarted;
    myController.ScreenContactContinued += MyController_ScreenContactContinued;
    myController.ScreenContactEnded += MyController_ScreenContactEnded;

    myController.RotationChanged += MyController_RotationChanged;
}
```

11. Add the event handler method for the **ScreenContactStarted** event. This method will create a **DialColorControl** if one doesn't already exist, and then place that control in the **DialCanvas** at the location of the Surface Dial. This lab uses a hardcoded value for the size of the control. You should determine if the **RadialControllerScreenContact.Bounds** property in the **RadialControllerScreenContactStartedEventArgs** parameter will provide what you need to size the visual you display on the screen.

```
private async void MyController_ScreenContactStarted(RadialController sender,
RadialControllerScreenContactStartedEventArgs args)
{
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
    {
        if (null == dialControl)
        {
            dialControl = new DialColorControl();
            DialCanvas.Children.Add(dialControl);
        }

        dialControl.Width = 300;
        dialControl.Height = 300;

        Canvas.SetLeft(dialControl, args.Contact.Position.X - 150);
        Canvas.SetTop(dialControl, args.Contact.Position.Y - 150);

        dialControl.Visibility = Visibility.Visible;
    });
}
```

12. In the **OnScreenVisualPage** class add the event handler for the **ScreenContactContinued** event. This will ensure the dialControl is placed on the **Canvas** in the correct location if the Surface Dial has moved on the screen.

```
private async void MyController_ScreenContactContinued(RadialController sender,
    RadialControllerScreenContactContinuedEventArgs args)
{
    if (null != dialControl)
    {
        await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
() =>
        {
            Canvas.SetLeft(dialControl, args.Contact.Position.X - 150);
            Canvas.SetTop(dialControl, args.Contact.Position.Y - 150);
        });
    }
}
```

13. Add the method to handle the **ScreenContentEnded** event. This will hide (collapse) the dialControl when the Surface Dial is removed from the screen.

```
private async void MyController_ScreenContactEnded(RadialController sender, object
args)
{
    if (null != dialControl)
```

```
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
            () =>
            {
                dialControl.Visibility = Visibility.Collapsed;
            });
        }
    }
```

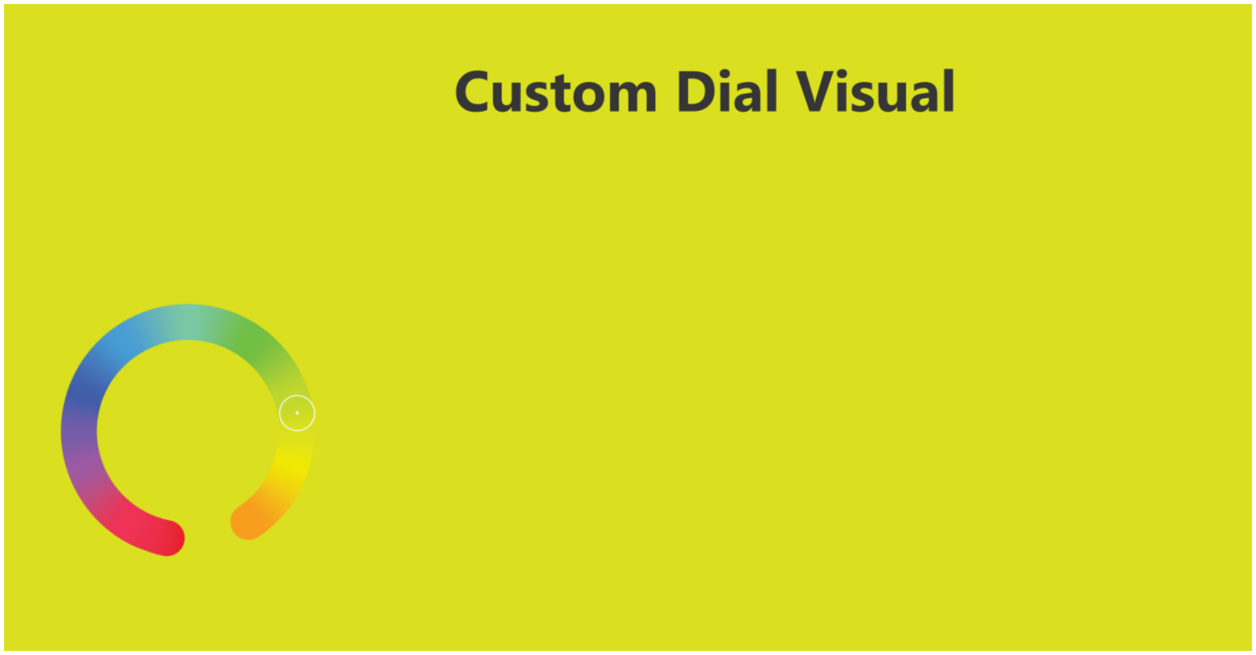
14. Implement the method to handle the **RotationChanged** event on the **RadialController**. This will set the rotation of the **DialColorControl** and retrieve the brush of the selected color, which is then used to set the background color.

```
private async void MyController_RotationChanged(RadialController sender,
RadialControllerRotationChangedEventArgs args)
{
    if (null != dialControl)
    {
        await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
        {
            dialControl.Rotation += args.RotationDeltaInDegrees;
            if (dialControl.Rotation > 0)
            {
                dialControl.Rotation = 0;
            }
            else if (dialControl.Rotation < -315)
            {
                dialControl.Rotation = -315;
            }
            MainGrid.Background = dialControl.ColorBrush;
        });
    }
}
```

15. Edit the **ColorMenuItem_Invoked** method to disable the **RadialControllerMenu** once the menu item is selected. This will hide the default menu so that our control is shown instead.

```
private void ColorMenuItem_Invoked(RadialControllerMenuItem sender, object args)
{
    Debug.WriteLine("Item invoked");
    myController.Menu.IsEnabled = false;
}
```

16. Build and run the application (**F5**). When you place the Surface Dial on the screen of the Surface Studio and click down on the Surface Dial button, you should see the color menu item. Select that menu item and invoke it (push down on the Surface Dial button). Then when you place the Surface Dial down on the screen somewhere else you will see the **DialColorControl** displayed on the screen around the Surface Dial. You can rotate the Surface Dial to select a different color for the background.

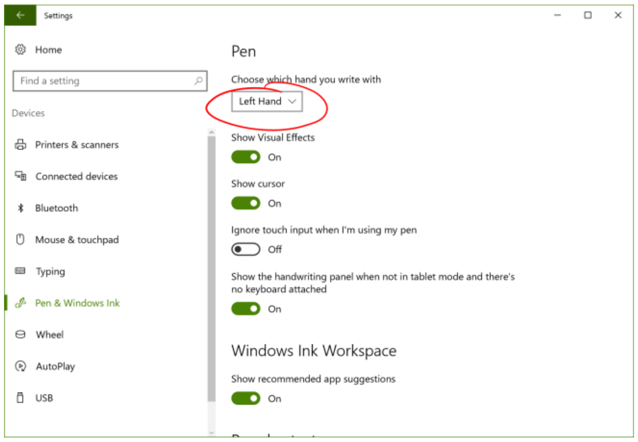


17. Open the **DialColorControl.xaml.cs** file in the **Controls** folder.

18. Review the constructor for the **DialColorControl** class again. You can see that the code checks for the **HandPreference** value in the **UISettings** object. If the customer is left handed the code rotates the control by 180 degrees.

```
public DialColorControl()
{
    this.InitializeComponent();
    //handedness for on-screen UI
    Windows.UI.ViewManagement.UISettings settings =
        new Windows.UI.ViewManagement.UISettings();
    rightHanded = (settings.HandPreference ==
        Windows.UI.ViewManagement.HandPreference.RightHanded);
    //if left handed then rotate the entire control 180
    if (!rightHanded)
    {
        RotateTransform trans = new RotateTransform();
        trans.Angle = 180;
        colorControl.RenderTransform = trans;
    }
}
```

19. You can test this by opening the Settings in Windows and changing the preference. Open the Windows Settings and navigate to **Devices | Pen & Windows Ink**. Change the setting and then test the app again.



Pen

Choose which hand you write with

[✓ Left Hand]

Show Visual Effects

On

Show cursor

On

Ignore touch input when I'm using my pen

Off

Show the handwriting panel when not in tablet mode and there's no keyboard attached

On

Windows Ink Workspace

Show recommended app suggestions

On

In this Lab you have learned that the Surface Dial device raises ScreenContact events when it placed on the screen, moved, and removed from the screen. You can use those events and the **RadialControllerScreenContact** properties in the event arguments to draw a custom control on the screen that can be controlled by the Surface Dial.

When you add your own control to the screen to interact with the Surface Dial you should always consider the **HandPreference** of the customer.