A Complex Custom Type Editor Example

This example demonstrates how to create a custom type that will be displayed in the Scene Composer property grid as custom editor. This example assumes the developer is familiar with WPF DataTemplating, DataBinding, CusotmControls, and DependencyProperties.

The example will create a custom control that will be used to edit a flags enum as a list of CheckBoxes. This example assumes you are already familiar with scene composer plugins and you are familiar with implementing a simple custom type editor as described in A Simple Custom Type Editor and enum custom types as described in A Custom Enum Type Editor

Providing a custom editing Data Template

Scene composer will invoke the GetDataTemplate method to request a WPF DataTemplate for editing the custom types. If this method returns null the default datatemplate will be used. If the method returns a DataTemplate then this custom template will be used. The class will load a resource dictionary that has the data template defined and will return a template with a specific key.

```
ResourceDictionary res = Application.LoadComponent(new Uri("/CustomTypeEditor;component/CustomC
public System.Windows.DataTemplate GetDataTemplate(string type)
{
   //We return the data template by name
   // See CustomComplexTypeEditorPluginUtils/DataTemplateDictionary.xam for more details on the
   return (DataTemplate)res["flagsTemplate"];
}
```

Define a custom editing Data Template

The data template is defined in a resource dictionary. The data template will be instantiated in the property grid on the right side where the property value is usually located. The template can contain any valid control. The passing of values between the template and the property grid is done using the DataContext of the DataTemplate The most important property of the DataContext is the Value property which represents the value of the property. This property is the only way for the DataTemplate to pass a value back to the property grid. Other read-only properties that the data template can use are:

- bool IsReadOnly specified weather the value should be read-only or not
- IEnumerable AvailableValues a list of the values that was returns by the GetAvailableValues of the TypeEditorPlugin
- CustomType - the name custom type name of the property
- Description - the description of the property extracted from metainfo
- Format - the format to be used for the property, also exacted from metainfo
- IsEnabled - !IsReadOnly, used for convenient binding instead of IsReadOnly
- IsError - If the conversion of the property fails, this property will be true. Can be used to show UI indicating the value is invalid
- IsUndefined - If there are more items selected in the property grid, and their values for the property is different this property will be true
- MaxValue - The double max value extracted from metainfo
- MinValue - The double min value extracted from metainfo
- Name - The name of the property extracted from metainfo.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:e="clr-namespace:CustomTypeEditor.CustomComplexTypeEditorPluginUt
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <DataTemplate x:Key="flagsTemplate">
      <e:CustomFlagsEditor Value="{Binding Value, Mode=TwoWay}"></e:CustomFlagsEditor>
   </DataTemplate>
</ResourceDictionary>
```

The datatemplate uses a custom control CustomFlagsEditor that is detailed in the code but not explained. The control is a WPF Custom control that uses standard data binding techniques to display a list of check boxes for editing the value.

The full code for the example:

CustomComplexTypeEditorPlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Controls;
using FTC.SDKInterface.Plugins;
using System.Windows;

namespace CustomTypeEditor
{
   /// <summary>
   /// This example demonstrates how to create a custom type that will be displayed in the Scene
   /// This example assumes the developer is familiar with WPF DataTemplating, DataBinding, Cusc
   ///
   /// The example will create a custom control that will be used to edit a flags enum as a list
   ///
   /// To use the example plug-in type copy the output of this project in the ($FTCInstallPath$)
   ///
   /// If you do not have a widget that uses this type, but with to test it woks in Scene Compos
```

```
/// This will override any other widgets and will load only the test widget that contains the
/// You can modify the WidgetMetaInfo.txt to include other custom types you wish to test.
/// </summary>
public class CustomComplexTypeEditorPlugin : ICustomTypeEditorExt
{

   /// <summary>
   /// The actual enum used for the values.
   /// </summary>
   [Flags]
   public enum CustomEnumFlags
   {
      EnumValue1 = 1,
      EnumValue2 = 1 << 2,
      EnumValue3 = 1 << 3
   };

   public string Name
   {
      get { return "Custom Type Editor"; }
   }

   public string Description
   {
      get { return "Sample custom type editor"; }
   }

   public string Version
   {
      get { return "1.1"; }
   }

   /// <summary>
   /// The name used for the enum. This will be used to identify the type in the WidgetMetaIr
   /// The name in the metainfo file should be prefixed with "custom://", to mark the fact th
   /// For this type the metainfo type name should be "custom://CustomFlagsEnumTypeName"
   /// </summary>
   private const string CustomFlagsEnumTypeName = "CustomFlagsEnumTypeName";

   /// <summary>
   /// Returns a list of types supported by the plug-in.
   /// </summary>
   public IEnumerable<string> SupportedTypes
   {
      get
      {
         yield return CustomComplexTypeEditorPlugin.CustomFlagsEnumTypeName;
      }
   }

   public void Initialize()
   {
   }
```

```
 /// <summary>
 /// Returns the value that will be written in the asset file.
 /// </summary>
 /// <param name="type">The type of the value.</param>
 /// <param name="value">The value</param>
 /// <returns>The string to be written in the asset file.</returns>
 public string GetAssetValue(string type, string value)
 {
    return value;
 }

/// <summary>
 /// Returns the default value for a type. This method will be used to initialize the value
 /// </summary>
 /// <param name="type">The type for which the default value is requested.</param>
 /// <returns>The default value</returns>
 public string GetDefaultInternalValue(string type)
 {
    switch (type)
    {
       default:
          throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.N

       case CustomComplexTypeEditorPlugin.CustomFlagsEnumTypeName:
          return "0";
    }
 }
 /// <summary>
 /// Tests if the value is valid for the type.
 /// </summary>
 /// <param name="type">The type of the value</param>
 /// <param name="value">The value to be tested for validity</param>
 /// <returns>Weather the value is valid for the type</returns>
 public bool IsInternalValueValid(string type, string value)
 {
    int intValue;

    switch (type)
    {
       default:
          throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.N

       case CustomComplexTypeEditorPlugin.CustomFlagsEnumTypeName:
         string[] tokens = value.Split(';');
         return tokens.Length == 2 && int.TryParse(tokens[0], out intValue);
    }
 }

/// <summary>
 /// Used to display a dialog that allows the user to edit a value of  a given type.
 ///
 /// This method is invoked when the user presses the edit button (usually with the text '.
```

```
        ///
        /// The dialog can be defined inside the plug-in module and will AUTOMATICALLY inherit the
        /// See the CustomEditor.xaml for more details.
        /// </summary>
        /// <param name="type">The type to be edited.</param>
        /// <param name="initialValue">The initial value of the type.</param>
        /// <param name="value">The new value that the user inputs</param>
        /// <returns>Weather the user confirmed the edit. (Aka weather the OK button was pressed)<
        public bool ShowEditValueDialog(string type, string initialValue, out string value)
        {
           value = initialValue;
           return false;
        }

        /// <summary>
        /// Returns the available values for a certain type.
        /// This method is usually use with simple enums or lists of values. In this example we wi
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public IEnumerable<CustomTypeDisplayValue> GetAvailableValues(string type)
        {
           if (type == CustomFlagsEnumTypeName)
           {
              return null;
           }
           else
           {
              throw new ArgumentException("Unsupported type '" + type + "' in plug-in " + this.Nam
           }
        }

        /// <summary>
        /// The resource dictionary that contains the data template to be used
        /// </summary>
        ResourceDictionary res = Application.LoadComponent(new Uri("/CustomTypeEditor;component/Cu

        /// <summary>
        /// Returns the data template used for editing the value.
        /// this example loads the data template from
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public System.Windows.DataTemplate GetDataTemplate(string type)
        {
           //We return the data template by name
           // See CustomComplexTypeEditorPluginUtils/DataTemplateDictionary.xam for more details c
           return (DataTemplate)res["flagsTemplate"];
        }

        /// <summary>
        /// Specified weather the type has a editor dialog.
        /// The editor dialog button is usually shown on the right side of the property grid field
```

```
      /// Since in this plug-in we want to use a custom control to show the enum value, the edit
      /// </summary>
      /// <param name="type"></param>
      /// <returns></returns>
      public bool HasShowEditValueDialog(string type)
      {
         return false;
      }
   }
}
```

DataTemplateDictionary.xaml

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:e="clr-namespace:CustomTypeEditor.CustomComplexTypeEditorPluginUtils"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <!--
   We create a data template for our custom flags enum.
   The data template will be instantiated in the property grid on the right side where the prope
   The template can contain any valid control. The passing of values between the template and th
   The most important property of the DataContext is the Value property which is bound here to t
   Other read-only properties are:
      - bool IsReadOnly specified weather the value should be read-only or not
      - IEnumerable AvailableValues  a list of the values that was returns by the GetAvailableVa
      - CustomType  - the name custom type name of the property
      - Description - the description of the property extracted from metainfo
      - Format - the format to be used for the property, also exacted from metainfo
      - IsEnabled - !IsReadOnly, used for convenient binding instead of IsReadOnly
      - IsError - If the conversion of the property fails, this property will be true. Can be us
      - IsUndefined - If there are more items selected in the property grid, and their values fo
      - MaxValue - The double max value extracted from metainfo
      - MinValue - The double min value extracted from metainfo
      - Name - The name of the property extracted from metainfo.
   -->
   <DataTemplate x:Key="flagsTemplate">
      <e:CustomFlagsEditor Value="{Binding Value, Mode=TwoWay}"></e:CustomFlagsEditor>
   </DataTemplate>
</ResourceDictionary>
```

CustomFlagsEditor.xaml

```
<UserControl x:Class="CustomTypeEditor.CustomComplexTypeEditorPluginUtils.CustomFlagsEditor"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
            mc:Ignorable="d"
            d:DesignHeight="300" d:DesignWidth="300">
    <Grid x:Name="Root">
      <ItemsControl ItemsSource="{Binding Items}">
         <ItemsControl.ItemTemplate>
            <DataTemplate>
               <CheckBox IsChecked="{Binding IsChecked}" Content="{Binding Name}"></CheckBox>
            </DataTemplate>
         </ItemsControl.ItemTemplate>
      </ItemsControl>
   </Grid>
</UserControl>
```

CustomFlagsEditor.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.ComponentModel;

namespace CustomTypeEditor.CustomComplexTypeEditorPluginUtils
{
   /// <summary>
   /// This is a custom control used for the actual value editing.
   /// The control has a Value property which is bound to the value property for the data contex
   /// When changing the value of the Value property it will be passed back in the property gric
   /// The control uses standard WPF methods to present a list with checkboxes, but the control
   /// </summary>
   public partial class CustomFlagsEditor : UserControl
   {
      public CustomFlagsEditor()
      {
         InitializeComponent();
         this.Root.DataContext = new CustomFlagsEditorDataContext(this);
      }



      public string Value
      {
```

```
        get { return (string)GetValue(ValueProperty); }
        set { SetValue(ValueProperty, value); }
    }

    // Using a DependencyProperty as the backing store for Value.  This enables animation, sty
    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value", typeof(string), typeof(CustomFlagsEditor), new UI
        {
            var editor = d as CustomFlagsEditor;
            ((CustomFlagsEditorDataContext)editor.Root.DataContext).UpdateValue();
        }));

}

 public class CustomFlagsEditorDataContext
 {
    public CustomFlagsEditorDataContext(CustomFlagsEditor editor)
    {
      this.Editor = editor;

       this.Items = (from e in Enum.GetValues(typeof(CustomComplexTypeEditorPlugin.CustomEnumF
                     select new CustomFlagsItem(this)
                     {
                        Name = e.ToString(),
                        Value = e
                     }).ToArray();
    }

    public void UpdateValue()
    {
       this.value = (CustomComplexTypeEditorPlugin.CustomEnumFlags)Enum.Parse(typeof(CustomCom
       foreach (var item in Items)
       {
          item.RaisePropertyChanged("IsChecked");
       }
   }

    public IEnumerable<CustomFlagsItem> Items { get; set; }

    public CustomFlagsEditor Editor { get; set; }

    CustomComplexTypeEditorPlugin.CustomEnumFlags value;
    public CustomComplexTypeEditorPlugin.CustomEnumFlags Value
    {
       get { return this.value; }
       set
       {
          if (this.value != value)
          {
             this.value = value;
            this.Editor.Value = ((int)value).ToString();
          }
       }
    }
```

```
  }

   public class CustomFlagsItem : INotifyPropertyChanged
   {
      private CustomFlagsEditorDataContext customFlagsEditorDataContext;

     public CustomFlagsItem(CustomFlagsEditorDataContext customFlagsEditorDataContext)
      {
         this.customFlagsEditorDataContext = customFlagsEditorDataContext;
     }

      public bool IsChecked
      {
         get { return (this.customFlagsEditorDataContext.Value & this.Value) != 0; }
         set
        {
            if (this.IsChecked != value)
            {
               if (value)
               {
                  this.customFlagsEditorDataContext.Value |= this.Value;
               }
               else
               {
                  this.customFlagsEditorDataContext.Value &= ~this.Value;
               }
               this.RaisePropertyChanged("IsChecked");
            }
         }
      }
      public string Name { get; set; }

      public CustomComplexTypeEditorPlugin.CustomEnumFlags Value { get; set; }

      public event PropertyChangedEventHandler  PropertyChanged;

      public void RaisePropertyChanged(string name)
      {
         if(this.PropertyChanged != null)
         {
            this.PropertyChanged(this, new PropertyChangedEventArgs(name));
         }
      }
   }
}
```

Revision #3
Created 2023-02-15 07:31:19 UTC by Tsuyoshi.Kato
Updated 2023-11-10 13:46:32 UTC by Elisabeth Zauner