A Custom Enum Type Editor

This example demonstrates how to create a custom enum type that will be displayed in the Scene Composer property grid as a combo of values.

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

Defining the plugin class

To implement advances custom type feature the plugin class must implement ICustomTypeEditorExt instead of ICustomTypeEditor Most of the methods are the same, just that this interface adds several other features we will use for this sample

```
public class CustomEnumTypeEditorPlugin : ICustomTypeEditorExt
{
}
```

The Enum

We define an actual C# enum for the values of our custom enum. The values of this enum will never reach Scene Composer, as the actual values for out custom type will always be strings, but for our sample it is more convenient to use the values from this enum as a basis for our custom type.

```
public enum CustomEnum
{
   EnumValue1,
   EnumValue2,
   EnumValue3
};
```

Available Values

The GetAvailableValues method must return the available values for the type. The values will be wrapped in a CustomTypeDisplayValue that holds both the actual value the custom type will have, but also a display string for the value.

```
public IEnumerable<CustomTypeDisplayValue> GetAvailableValues(string type)
{
   if (type == CustomEnumTypeName)
   {
      return from v in Enum.GetValues(typeof(CustomEnum)).OfType<CustomEnum>()
             select new CustomTypeDisplayValue()
             {
                DisplayName = v.ToString(),
                Value = ((int)v).ToString()
             };
   }
   else
   {
      throw new ArgumentException("Unsupported type '" + type + "' in plug-in " + this.Name);
   }
}
```

Implementing other interface methods.

As with the simple custom type example (A Simple Custom Type Editor) we must implement the methods for offering the default value, value validation, and asset value. The full code might be seen on the bottom of the page.

The other method of the ICustomTypeEditorExt, GetDataTemplate, interface will be explained in the A Complex Custom Type Editor Example example. For now the method will simply return null defaulting to Scene Composer standard handling for custom enums, which is to display a ComboBox with the values.

```
public System.Windows.DataTemplate GetDataTemplate(string type)
{
   return null;
}
```

The full code for the example:

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

namespace CustomTypeEditor
{
   /// <summary>
   /// This example demonstrates how to create a custom enum type that will be displayed in the
   ///
   /// 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 you want to test it woks in Scene Co
/// 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 CustomEnumTypeEditorPlugin : ICustomTypeEditorExt
{

   /// <summary>
   /// The actual enum used for the values.
   /// Since the enum is never actually exposed to Scene Composer and the values stored as st
   /// </summary>
   public enum CustomEnum
   {
      EnumValue1,
      EnumValue2,
      EnumValue3
   };

   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 WidgetMetaIn
   /// 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://CustomEnumTypeName"
   /// </summary>
   private const string CustomEnumTypeName = "CustomEnumTypeName";

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

   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 CustomEnumTypeEditorPlugin.CustomEnumTypeName:
             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 CustomEnumTypeEditorPlugin.CustomEnumTypeName:
             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.
        /// If the enumeration is not null then a combo box will be used for editing this type.
        /// This method is intended to be used with custom enums or lists of values.
        /// In this case the method returns the values of the CustomEnum type.
        /// The values could be taken from any other source provided the ICustomTypeEditor knows h
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public IEnumerable<CustomTypeDisplayValue> GetAvailableValues(string type)
        {
           if (type == CustomEnumTypeName)
           {
              return from v in Enum.GetValues(typeof(CustomEnum)).OfType<CustomEnum>()
                     select new CustomTypeDisplayValue()
                     {
                        DisplayName = v.ToString(),
                        Value = ((int)v).ToString()
                     };
           }
           else
           {
              throw new ArgumentException("Unsupported type '" + type + "' in plug-in " + this.Nam
           }
        }

        /// <summary>
        /// Returns the data template used for editing the value.
        /// In this example we will use the default custom enum behavior in scene composer so we w
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public System.Windows.DataTemplate GetDataTemplate(string type)
        {
           return null;
        }

        /// <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 combo box to show the enum values the editor di
      /// </summary>
      /// <param name="type"></param>
      /// <returns></returns>
      public bool HasShowEditValueDialog(string type)
      {
        return false;
      }
   }
}
```

Revision #3
Created 2023-02-15 07:32:26 UTC by Tsuyoshi.Kato
Updated 2023-11-10 13:47:09 UTC by Elisabeth Zauner