Type Editor Plugins

- A Complex Custom Type Editor Example
- A Custom Enum Type Editor
- A Simple Custom Type Editor
- Widget Metainfo Example

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));
         }
      }
   }
}
```

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;
      }
   }
}
```

A Simple Custom Type Editor

Creating a custom type editor

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll. The resulting dll from your project (along with any non .NET dependecies) must be placed in the Plugins directory in the FTC install location.

The Plugin class.

The first step is to create a class that implements the ICustomTypeEditor interface. This is the type that will be instantiated by Scene Composer whose methods will be called to handel custom type specific actions.

```
public class CustomTypeEditorPlugin : ICustomTypeEditor
{
}
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

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

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

public string Version
{
   get { return "1.1"; }
}
public void Initialize(IUnityContainer unityContainer)
{
}
```

Specifying what types are supported

We implement the SupportedTypes member of the ICustomTypeEditor. This property defines the name of the custom types supported by the plugin. These names will be passed back to the plugin wherever the type must be identified. Also these are the names that should be used in the meta

info file.

```
private const string CustomTypeName = "CustomTypeEditor";
private const string AnotherCustomTypeName = "AnotherCustomTypeEditor";

public IEnumerable<string> SupportedTypes
{
   get
   {
      yield return CustomTypeEditorPlugin.CustomTypeName;
      yield return CustomTypeEditorPlugin.AnotherCustomTypeName;
   }
}
```

Defining the values of the type

Next we implement value related methods.
The GetDefaultInternalValue will return the default value for a type, for our types the defaults are "0;" and "0". The first custom type is a complex type that has two components separated by ';' the first component must be an integer, while the second will be a simple string. The second custom type is a simple integer value.

```
public string GetDefaultInternalValue(string type)
{
   switch (type)
   {
      default:
         throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.Name
);

      case CustomTypeEditorPlugin.CustomTypeName:
         return "0;";

      case CustomTypeEditorPlugin.AnotherCustomTypeName:
         return "0";
   }
}
```

The IsInternalValueValid method validates whether the value passed as argument is valid for the type.

```
public bool IsInternalValueValid(string type, string value)
{
   int intValue;

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

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

        case CustomTypeEditorPlugin.AnotherCustomTypeName:
            return int.TryParse(value, out intValue);
    }
 }
```

The GetAssetValue converts the internally stored value for a type to an asset compatible value. This value is dependant on the consumer of the property. For our example we will use the same strig as the internal value.

```
 public string GetAssetValue(string type, string value)
 {
    switch (type)
    {
        default:
            throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.Name
);

        case CustomTypeEditorPlugin.CustomTypeName:
            return value;

        case CustomTypeEditorPlugin.AnotherCustomTypeName:
            return value;
    }
 }
```

Last we must define the ShowEditValueDialog method that is used for the actual value editing. We assume the existence of CustomEditor dialog window which will be detailed later, that has two textboxes that out custom editor can access.
The method calls the dialog, and collects the values from the textboxes. If the user hits ok in the dialog then the method will return true, otherwise it will return false, and the new value will not be saved.

```
 public bool ShowEditValueDialog(string type, string initialValue, out string value)
 {
    int intValue;
    var dlg = new CustomEditor();

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

        case CustomTypeEditorPlugin.CustomTypeName:
            string[] tokens = initialValue.Split(';');
            if (tokens.Length == 2)
```

```
         {
            if (int.TryParse(tokens[0], out intValue))
            {
               dlg.textBox0.Text = tokens[0];
            }
            dlg.textBox1.Text = tokens[1];
         }

         if ((bool)dlg.ShowDialog())
         {
            string newValue = dlg.textBox0.Text + ";" + dlg.textBox1.Text;
            if (this.IsInternalValueValid(type, newValue))
            {
               value = newValue;
               return true;
            }
         }
         break;

      case CustomTypeEditorPlugin.AnotherCustomTypeName:
         if (int.TryParse(initialValue, out intValue))
         {
            dlg.textBox0.Text = initialValue;
         }

         dlg.textBox1.Visibility = System.Windows.Visibility.Collapsed;

         if ((bool)dlg.ShowDialog() && this.IsInternalValueValid(type, dlg.textBox0.Text))
         {
            value = dlg.textBox0.Text;
            return true;
         }
         break;
   }

   value = initialValue;
   return false;
}
```

Defining the Dialog Window

Implicitly any dialog show in scene composer will use the same look and feel for the controls. For the window however, an explicit style must be set to give it the Scene Composer look and feel, that style is named 'DialogWindow'

```
<Window x:Class="CustomTypeEditor.CustomEditor"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Style="{DynamicResource DialogWindow}" ...>
     ...
```

The full code for the example:

CustomTypeEditorPlugin.cs

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

namespace CustomTypeEditor
{
   public class CustomTypeEditorPlugin : ICustomTypeEditor
   {
      public string Name
      {
         get { return "Custom Type Editor"; }
      }

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

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

      private const string CustomTypeName = "CustomTypeEditor";
      private const string AnotherCustomTypeName = "AnotherCustomTypeEditor";

      public IEnumerable<string> SupportedTypes
      {
         get
         {
            yield return CustomTypeEditorPlugin.CustomTypeName;
            yield return CustomTypeEditorPlugin.AnotherCustomTypeName;
         }
      }

      public void Initialize()
      {
      }

     public string GetAssetValue(string type, string value)
     {
        switch (type)
        {
           default:
              throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.
```

```
Name);

            case CustomTypeEditorPlugin.CustomTypeName:
               return value;

            case CustomTypeEditorPlugin.AnotherCustomTypeName:
               return value;
         }
      }

       public string GetDefaultInternalValue(string type)
       {
          switch (type)
          {
             default:
                throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.
Name);

             case CustomTypeEditorPlugin.CustomTypeName:
                return "0;";

             case CustomTypeEditorPlugin.AnotherCustomTypeName:
                return "0";
          }
       }

       public bool IsInternalValueValid(string type, string value)
       {
          int intValue;

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

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

             case CustomTypeEditorPlugin.AnotherCustomTypeName:
                return int.TryParse(value, out intValue);
          }
       }

       public bool ShowEditValueDialog(string type, string initialValue, out string value)
       {
          int intValue;
          var dlg = new CustomEditor();

          switch (type)
          {
```

```
            default:
               throw new ArgumentException("Unsupported type '" + type + "' in plugin " + this.
Name);

            case CustomTypeEditorPlugin.CustomTypeName:
               string[] tokens = initialValue.Split(';');
               if (tokens.Length == 2)
               {
                  if (int.TryParse(tokens[0], out intValue))
                  {
                     dlg.textBox0.Text = tokens[0];
                  }
                  dlg.textBox1.Text = tokens[1];
               }

               if ((bool)dlg.ShowDialog())
               {
                  string newValue = dlg.textBox0.Text + ";" + dlg.textBox1.Text;
                  if (this.IsInternalValueValid(type, newValue))
                  {
                     value = newValue;
                     return true;
                  }
               }
               break;

            case CustomTypeEditorPlugin.AnotherCustomTypeName:
               if (int.TryParse(initialValue, out intValue))
               {
                  dlg.textBox0.Text = initialValue;
               }

               dlg.textBox1.Visibility = System.Windows.Visibility.Collapsed;

               if ((bool)dlg.ShowDialog() && this.IsInternalValueValid(type, dlg.textBox0.Text)
               {
                  value = dlg.textBox0.Text;
                  return true;
               }
               break;
         }

         value = initialValue;
         return false;
      }
   }
}
```

CustomEditor.xaml

```
<!--
   The resources for controls are implicitly inherited from the application, so all the controls
```

```
   To have a dialog window that looks the same as Scene Composer's you must specify the DialogWi
   Other useful resource keys are :
      - NormalBrush - Used for the background of some items
      - LightBrush - Used for the background of some items
      - TextBrush - Used for the text color.
      - NormalBorderBrush - The normal border color of a control
      - HoverBrush - used for highlighting the hover over an element
      - ControlBackgroundBrush - used for the background of a control
      - SelectedBackgroundBrush - used to highlight a selected element
      - WindowBackgroundBrush - used for the color of the window background
   -->
<Window x:Class="CustomTypeEditor.CustomEditor"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Style="{DynamicResource DialogWindow}"
    Title="Custom Type Editor" Height="172" Width="249" KeyDown="OnKeyDown" ResizeMode="NoResize

    <Grid>

      <Button Margin="34,99,0,0" Name="button1" HorizontalAlignment="Left" Width="76" Click="OnC
      <Button HorizontalAlignment="Right" Margin="0,99,21,0" Name="button2" Width="75" Click="On
      <TextBlock Height="25" HorizontalAlignment="Left" Margin="12,33,0,0" Name="label0" Vertica
      <TextBlock Height="25" Margin="12,64,0,0" Name="label1" VerticalAlignment="Top" Horizontal
      <TextBlock Height="25" Margin="12,-1,0,0" Name="label4" VerticalAlignment="Top" Horizontal
      <TextBox Height="25" Margin="86,33,0,0" Name="textBox0" VerticalAlignment="Top" Horizontal
      <TextBox Margin="86,64,0,0" Name="textBox1" Height="25" VerticalAlignment="Top" Horizontal
   </Grid>
</Window>
```

CustomEditor.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;

namespace CustomTypeEditor
{
   public partial class CustomEditor : Window
   {
      public CustomEditor()
      {
         InitializeComponent();
```

```
        }

        private void OnOkButtonClick(object sender, RoutedEventArgs e)
        {
            this.DialogResult = true;
            this.Close();
        }

        private void OnCancelButtonClick(object sender, RoutedEventArgs e)
        {
            this.Close();
        }

        private void OnKeyDown(object sender, KeyEventArgs e)
        {
            switch (e.Key)
            {
                case Key.Escape:
                    this.Close();
                    return;

                case Key.Enter:
                    this.OnOkButtonClick(this, new RoutedEventArgs());
                    return;
            }
        }
    }
}
```

Widget Metainfo Example

This file is an example of a metainfo file that defines a widget. This file can be used to test the functionally of custom types. To do this you must copy the content below in a file called WidgetMetaInfo.txt and place it in the directory where Scene Composer is installed. This will replace the existing widget set with the widget defined in this file.
You can the replace the custom:// entries with your own custom types.

```
WidgetCount = 1
--------------------
  WidgetType = Widget3D
  Requirements =
  Name = Test3DWidget
  ReadableName = Test3DWidget
  Description = Test3DWidget. Used to test the functionality of custom types in the sample custo
  Category = Sample
  PropertyCount = 4
  --------------------
    Name = CustomFlags
    ReadableName = Custom Flags
    Description = An example of a custom type that uses a custom control to edit the value.
    Category = Sample
    Type = custom://CustomFlagsEnumTypeName
    RangeMin =
    RangeMax =
    DefaultValue =
    IsChangeable = 1
    --------------------
    Name = CustomEnum
    ReadableName = Custom Enum
    Description = An example of a custom type that uses the default enum behavior in Scene Compo
    Category = Sample
    Type = custom://CustomEnumTypeName
    RangeMin =
    RangeMax =
    DefaultValue =
    IsChangeable = 1
    --------------------
    Name = CustomType
    ReadableName = Custom Type
    Description = An example of a custom type that uses a dialog for editing
    Category = Sample
    Type = custom://CustomTypeEditor
    RangeMin =
    RangeMax =
    DefaultValue =
    IsChangeable = 1
    --------------------
    Name = AnotherCustomTypeEditor
```

```
ReadableName = Another Custom Type
Description = An example of another custom type that uses a dialog for editing
Category = Sample
Type = custom://AnotherCustomTypeEditor
RangeMin =
RangeMax =
DefaultValue =
IsChangeable = 1
--------------------
Name = AssociatedNode
ReadableName = AssociatedNode
Description = The associated root node associated to the widget
Category = Sample
Type = builtin://NodeEditor
RangeMin =
RangeMax =
DefaultValue =
IsChangeable = 1
--------------------
Name = Name
ReadableName = Name
Description = The name of the widget instance
Category = Sample
Type = builtin://StringEditor
RangeMin =
RangeMax =
DefaultValue =
IsChangeable = 1
--------------------
```