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;
           }
        }
     }
  }
```

Revision #3
Created 2023-02-15 07:33:19 UTC by Tsuyoshi.Kato
Updated 2026-07-17 08:13:27 UTC by Georg Stöbich