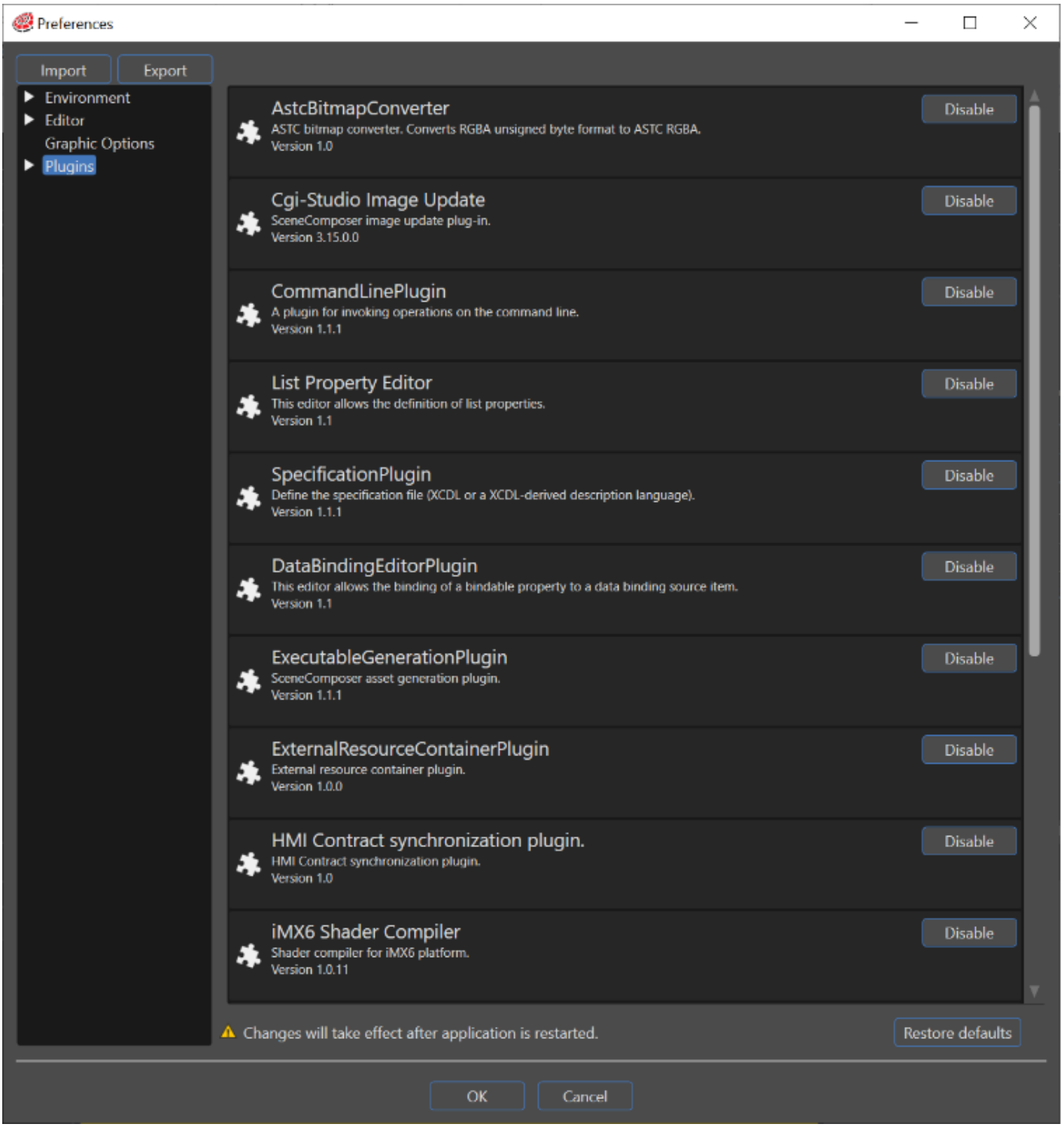
Extending SceneComposer

- Plugins Versions
- Plugins Configuration
- Custom Shader Compiler Plugin
- Bitmap Converter
- Executable Generation
- Type Editor Plugins
    - A Complex Custom Type Editor Example
    - A Custom Enum Type Editor
    - A Simple Custom Type Editor
    - Widget Metainfo Example
- UI Examples
    - A simple configurable plugin
    - A plugins that shows a custom dialog
    - A plugins that defines a custom menu
    - A simple plugin that consumes events
- Custom Properties Support
- FTC Command (Optional)
- Automation API
- Scripting
- Easy Mode Generator

Plugins Versions

Plugins Configuration

The version of every available plugin is displayed in "Preferences" > "Plugins" under the name of the plugin (the right side of the panel).



Plugins Configuration

Plugins Configuration

On the previous SC version just one type of importers was available (native importers using the mff framework). They were located in the folder ".\Importers". All the other plugins were managed assemblies located in the folder ".\Plugins". During start-up SC was trying to load all dll's from ".\Plugins" as managed plugins even if they were native dll's.

From the current 2.9 version of SceneComposer there will be three types of importer plugins:

- native/mff
- native/scx
- managed/scx

To keep the things well organized, the best options is to have all the plugins in one place. The Plugins.xml fileway allows to distinguish if it is native/managed and for native if it is mff(old)/scx(new).

```
<?xml version="1.0" encoding="utf-8"?>
<Plugins>
<Plugin FileName="EmeraldShaderCompiler.dll"/>
<Importer FileName="MffImporter.dll" IsManaged="false" API="Mff"/>
<Importer FileName="ScxImporter.dll" IsManaged="false" API="Scx"/>
</Plugins>
```

If this file is missing the native/mff importers will loaded from ".\Importers" and the managed plugins will be loaded from the folder ".\Plugins" as it was in the previous versions. If the configuration file is present, only the specified plugins will be loaded. A sample configuration file is delivered with SceneComposer.

Refer also to Starting SceneComposer.

Custom Shader Compiler Plugin

Custom Shader Compiler Plugin

SceneComposer SDK supplies basic means to create custom shader compilers plugins. A shader compiler plugin is a SceneComposer plugin which implements the IShaderCompiler interface. The IShaderCompiler interface is pretty simple and has only 3 methods to implement:

```
□string SupportedType { get; }
```

Specifies how this shader compiler can be referred by a platform. For example; if SupportedType is "DEVICEShaderCompiler" this shader compiler can be referred using the string "custom://DEVICEShaderCompiler". (DEVICE stands for your specific device)

```
□void AssetCompileShaderPair(string fragmentShaderFile,
□□□□□□□□□string vertexShaderFile,
□□□□□□□□□out byte[] compiledFragmentShader,
□□□□□□□□□out byte[] compiledVertexShader,
□□□□□□□□□out IEnumerable<ShaderCompilerMessage> compilerMessages);
```

This method is invoked by SceneComposer when the shaders are to be compiled at the time the asset file is generated for the target system. It supplies the full name of the shader frag and vert files and must return two buffers containing the compiled frag and vert shaders. Also, a list of compiler messages can be returned which are appended into the shader compiler output window to signal for example errors or other critical conditions.

The last method is CheckShaderPair which is called to verify the shaders under the following conditions:

- when the user invokes the "Compile Shader" option during editing. The user can invoke this option for a fragment shader, a vertex shader or a shader pair (shader program) containing both a fragment and vertex shader.
- when the asset file for the simulation environment is generated.
- when SceneComposer render data is updated.

```
□void CheckShaderPair(string fragmentShaderFile,
□□□□□□□□string vertexShaderFile,
□□□□□□□□out IEnumerable<ShaderCompilerMessage> compilerMessages);
```

Once all three methods are implemented and placing the resulting assembly into SceneComposer's plugin folder, the custom shader compiler will be available next time SceneComposer is started.

Bitmap Converter

Bitmap Converter

SceneComposer SDK supplies basic means to create custom bitmap converter plugins. A bitmap converter plugin is a SceneComposer plugin which implements the IBitmapConverter interface.

The IBitmapConverter interface is pretty simple:

IEnumerable<CustomFormat> SupportedTypes { get; }

Specifies the list of all custom formats supported by this converter which will be identified in SceneComposer by their name.

void Convert(BinaryReader reader, BinaryWriter writer, IBitmap bitmap, int customFormatId, bool shouldBeVerticallyFlipped, out int assetFormat, out int assetType, out bool assetIsVerticallyFlipped)

Converts an imported bitmap into a specific custom format using a format id and a flag which specifies the layout of the output data. This method is called when the asset or the render data is generated.

Please note that the integer values for asset format and asset type have to be defined in conjunction with the hardware platform capabilities. For example, the default asset type defines the format like byte, short, short565, short4444 or short 5551 as defined in Candera, and the asset format defines the bitmap format like this: public enum BitmapFormat { Rgb = 0, Rgba = 1, Alpha = 2, Luminance = 3, LuminanceAlpha = 4, Depth = 5, Coverage = 6, }

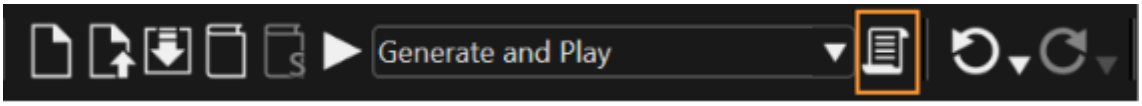
Once the interface is implemented and the resulting assembly is placed into SceneComposer's plugin folder, the custom bitmap converter will be available next time SceneComposer is started.

Executable Generation

Executable Generation

In the toolbar is available the "Automation Scripts" plug-in which provides the option to generate the asset files and run a script with certain commands defined by the user.

> The script will run when the asset will be generated.



The configuration dialog can be accessed via menu "File" > "Automation Scripts...", too.

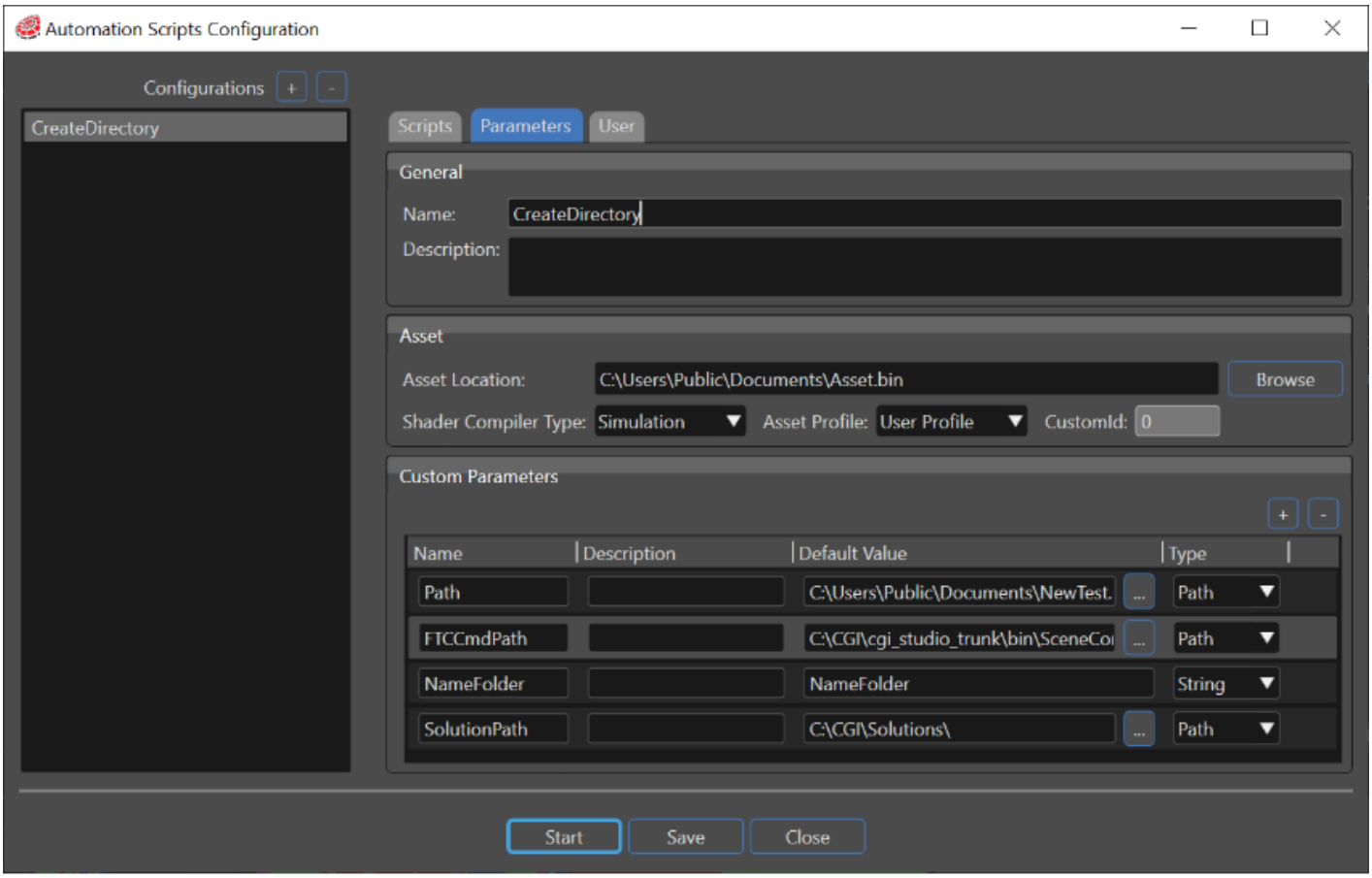
The user is able to define as many configurations as necessary, each configuration containing a set of default parameters, a set of custom parameters and a script to be run after a successful asset generation. The script is not mandatory and it can contain references to the default and custom parameters defined by the user.

> The set of custom parameters are specific for each solution and are saved in

```
USER_DIR\ AppData\Roaming\FTC.Launcher\v3.2.0.0\Configs\.ExecutableGeneration\
```

or

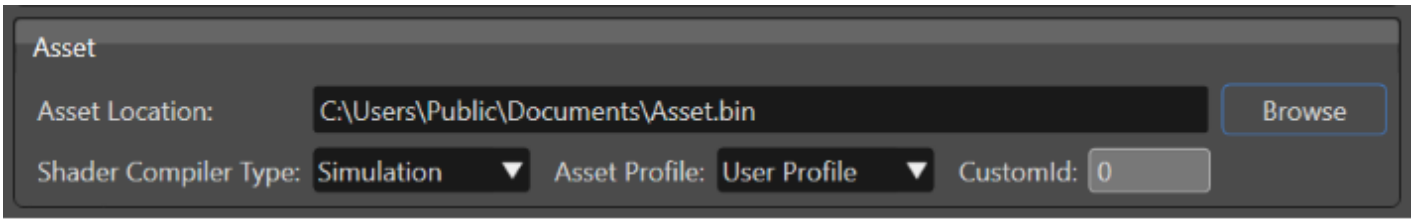
```
USER_DIR\ AppData\Roaming\SceneComposer\v3.2.0.0\Configs\.ExecutableGeneration\
```



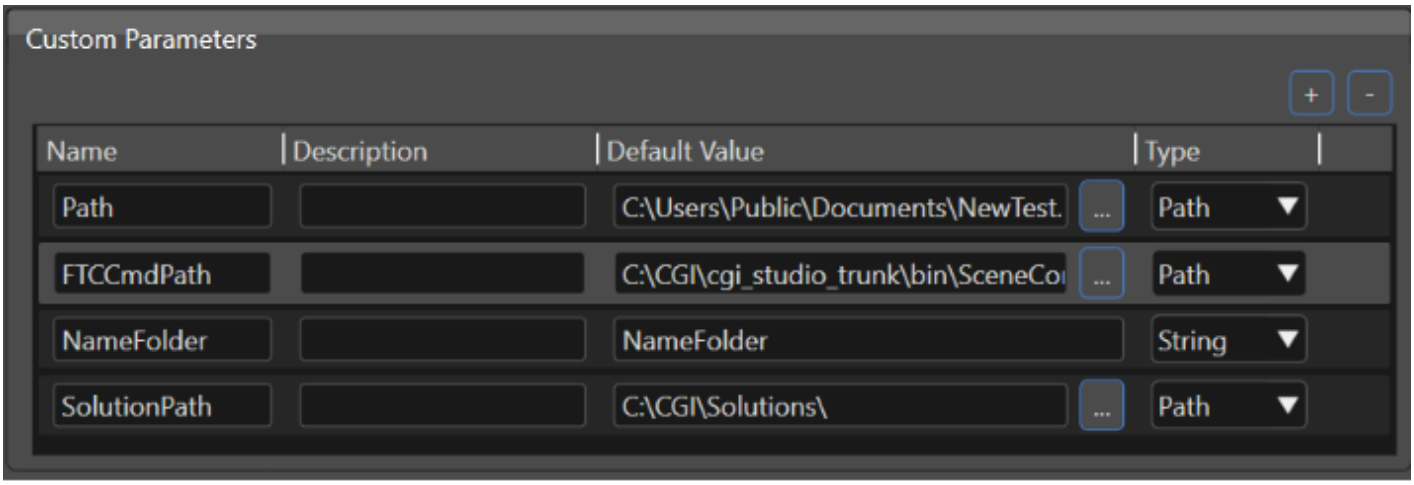
The **default parameters** names are predefined and they must be referred using the following keywords:

- $AssetLocation$ for the asset file location (in the image below "Asset Location")
- $AssetType$ for the asset type (in the image below "Shader Compiler Type")
- $AssetConfig$ for the asset configuration (in the image below "Asset Profiles")
- $AssetId$ for the asset id (in the image below "CustomId")

The default parameters will be referred using the custom parameter name between two "$" characters.



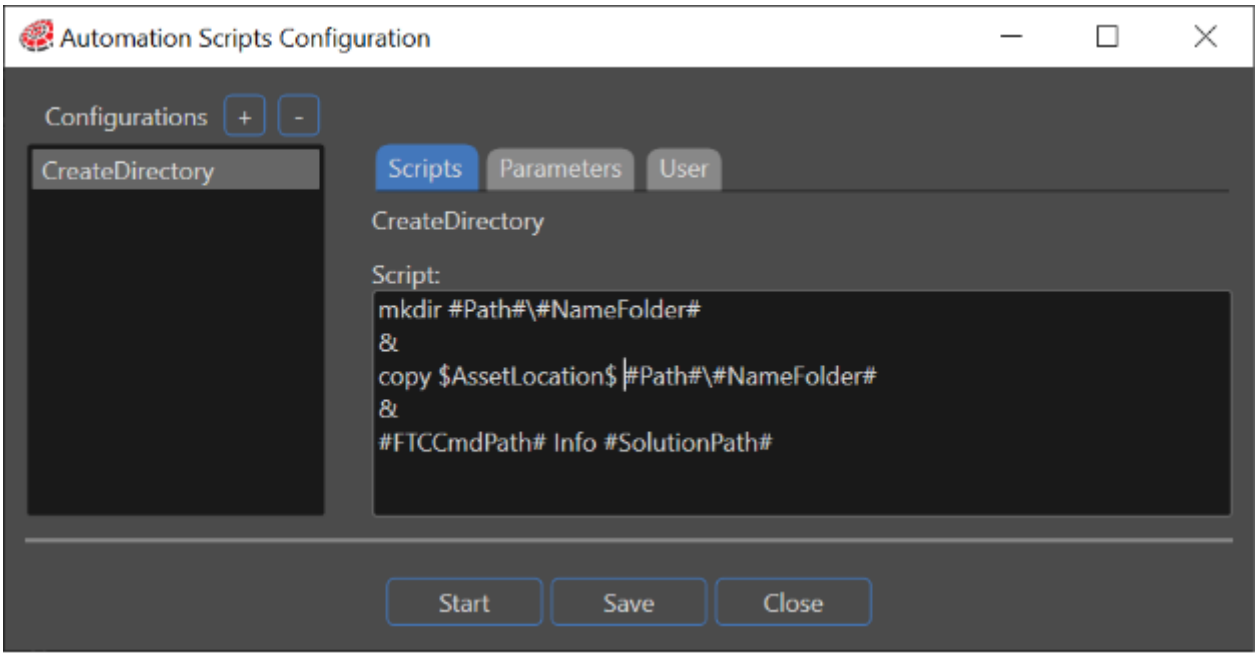
The custom parameters will be referred using the custom parameter name between two "#" characters.



Example:

```
#ParameterName#
```

When the script will be executed the parameters' names will be replaced with the values defined by the user.

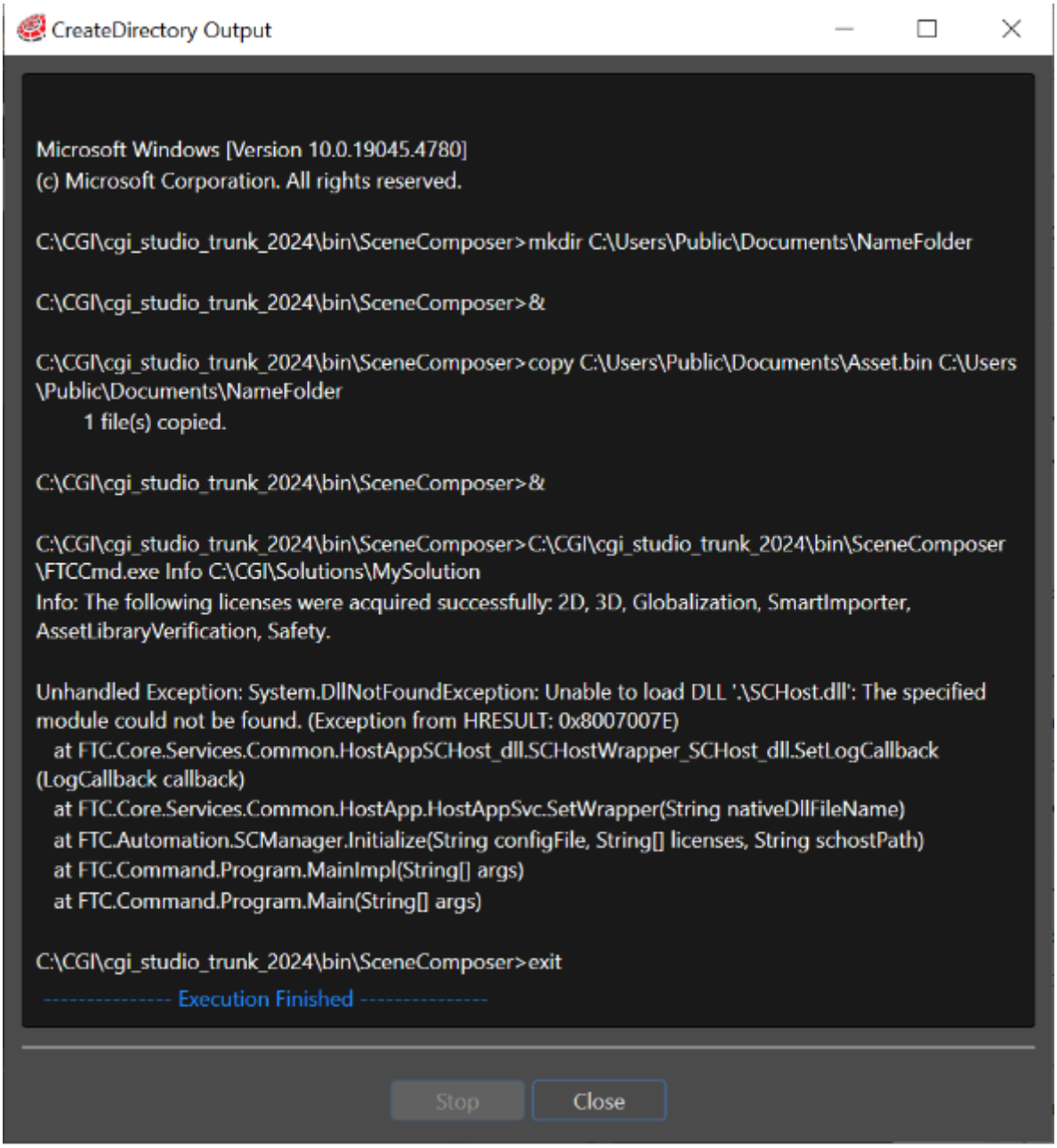


If script contains multiple commands they must be separated using the "&" character.

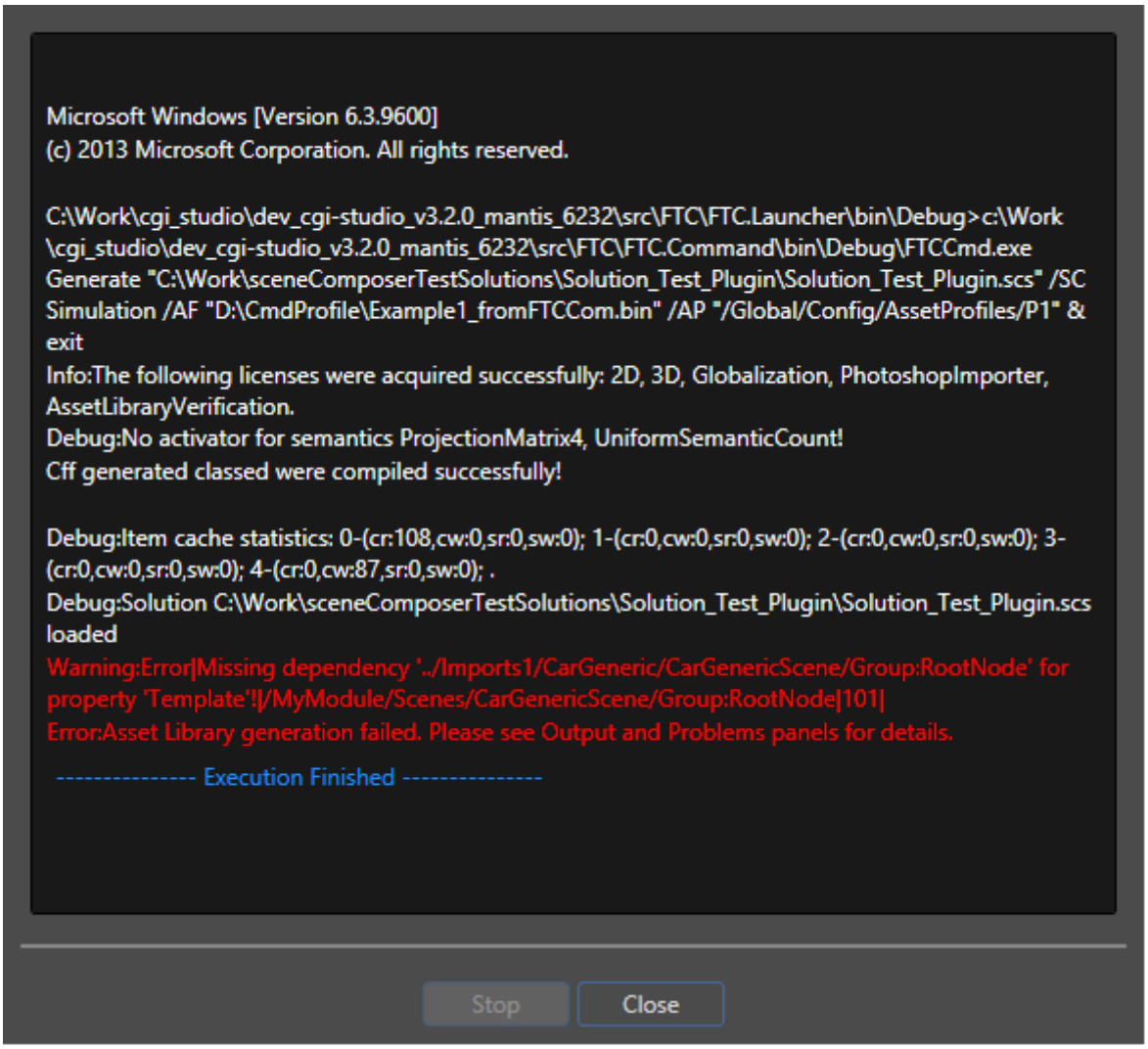
> As can be noticed from all the examples mentioned above, the "$", "#", and "&" characters are reserved. They cannot be used as values defined by the user.

Before the asset generation if a script is defined it will be verified if it is valid; if the script fails the validation the execution will be canceled (both: the asset generation and the commands execution

from the script), and the user will be notified with a message containing the parameters names mismatch.



When the script is executed a window will be opened in which will be logged the output generated by the execution of the commands. The execution can be suspended by clicking on the "Stop" button which is enabled only during script execution. The user also will be notified when all the commands are finished to be run.



If there are errors which do not allow to finish properly the execution of the script, they will be displayed - as can be seen in the image above - with red letters.

The user has the possibility to save the defined configurations by clicking on the "Save" button which is enabled when all the required fields are passing the validation. Also the "Start" button obey the same validation rules and it will be enabled only after all the validations will be successfully passed.

When the user will open the "Configuration" settings window it will be populated with the previously saved configurations.

Type Editor Plugins

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
--------------------
```

UI Examples

A simple configurable plugin

Creating a plugin that registers a configuration and uses the default configuration editor.

The Plugin class.

The first step is to create a class that implements the IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
public class SimpleConfigurablePlugin : IPluginService
{
}
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return return this.GetType().Name; }
}

public string Description
{
   get { return "A plugin that has configuration"; }
}

public string Version
{
  get { return "1.1.1"; }
}

public IEnumerable<int> FeatureIds
{
  get { return null; }
}
```

The Initialize method

First we need to register a custom cinfiguration interface and register to recieve an event when the configuration is loaded.

```
   ...
   var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface<I
```

```
  loader.ConfigLoaded += loader_ConfigLoaded;
  ...
```

Then we need to registter a defualt configuration definition of our custom type with the UI definition manager

```
unityContainer.Resolve<UIDefinitionManager>().RegisterUIElement
(
  new DefaultConfigDefinition<ISimpleConfiguration>(category: OptionCategories.Plugins, name: "
);
```

The full code for the example with extra info:

SimpleConfigurablePlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.UIComposition.UIConfiguration;
using FTC.SDKInterface.Plugins;
using FTC.SDKInterface.Services;
using Feat.UIComposition.Config;
using System.ComponentModel;
using Feat.UIComposition;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Menus;
using FTC.SDKInterface.Constants;
using Feat.Metadata.PropertyGrid;

namespace UIPlugins
{

   /// <summary>
   /// This examples demos a simple plugin that registers a configuration with FTC
   /// and uses the default configuration editor in scene composer.
   /// </summary>
   public class SimpleConfigurablePlugin : IPluginService
   {
      public string Description
      {
         get { return "A plugin that has configuration"; }
      }
      public string Name
      {
         get { return this.GetType().Name; }
      }

      public string Version
      {
         get { return "1.1.1"; }
      }
```

```
   public IEnumerable<int> FeatureIds
   {
      get { return null; }
   }

   public void Initialize(IUnityContainer unityContainer)
   {
      // We register the configuration interface with the configuration manager.
      // The object that is returned allow access to the config instance, allows the saving
      // and allows the registration of events pertinent to the configuration, such as the
      // Explicit loading or saving of the configuration is not necessary, as FTC will load
      // it is accessed, and will save it when the application closes.
      // The plugin developer can however explicitly save the configuration or reload it fr
      var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInt

      //We register to receive events when the configuration is loaded.
      loader.ConfigLoaded += loader_ConfigLoaded;


      // We register with the ui definition manager a default configuration definition with
      // We also specify the category in which to place the configuration, and the name to
      unityContainer.Resolve<UIDefinitionManager>().RegisterUIElement
      (
         new DefaultConfigDefinition<ISimpleConfiguration>(category: OptionCategories.Plugi
      );
   }

   /// <summary>
   /// Event handler for when the configuarion is loaded.
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   void loader_ConfigLoaded(object sender, ConfigLoadedEventArgs<ISimpleConfiguration> e)
   {
      // If the configuration was not loaded successfully, either this is the first time it
      // and there was no configuration to load, or the configuration file was corrupted.
      // Either way we initialize complex configuration properties, that could not be initi
      if (!e.ConfigLoadedSuccessfully)
      {
         e.Instance.OtherParameters = new List<string>
         {
            "Param1 "
         };
      }
   }


}

/// <summary>
/// The configuration interface. This interface will be implemented by Scene Composer and c
/// when it is registered with the Configuration manager service.
/// </summary>
```

```
    /// All properties declared  in the configuration interface must conform to standard .NET X
    ///
    /// If we use the default editor for configurations then we can control how the property gr
    ///
    /// The CategoryDisplayInfo attribute specifies that the default category should have no he
    ///
    /// The DefaultValue attribute specifies the default value to be used for the property, eit
    /// when the configuration is first created. \n
    ///
    /// The Category attribute dictates the category in which the property grid will display th
    ///
    /// The DisplayName attribute specifies what name the property grid should use for this pro
    /// Scene composer will use the actual property name.\n
    ///
    /// The Browsable attribute allows the hiding of properties in the property grid.
    [CategoryDisplayInfo("", StyleKey = PropertyGridConstants.NoHeaderCategory)]
    public interface ISimpleConfiguration : IConfiguration
    {

       [DefaultValue("format C:")]
       string CommandToExecute { get; set; }

       [DefaultValue(true)]
       [DisplayName("Show confirmations")]
       [Category("Options")]
       bool Confirm { get; set; }

       [DefaultValue(false)]
       [Category("Options")]
       bool Silent { get; set; }

       [Browsable(false)]
       List<string> OtherParameters { get; set; }
    }
}
```

A plugins that shows a custom dialog

Creating a plugin that displays a custom dialog

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll, Feat.UIComposition.dll and Feat.Core.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 IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
public class PluginWithMenuAndDialogPlugin : IPluginService
{

}
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Description
{
  get { return "A plugin with a menu item and dialog"; }
}

public string Name
{
  get { return this.GetType().Name; }
}

public string Version
{
  get { return "1.1.1"; }
}

public IEnumerable<int> FeatureIds
{
   get { return null; }
```

```
}
```

The initliaze method will register a menu command and a custom dialog Registering Commands: To register a command first an interface that derives from ICommandContainer must be created. The interface will contain properties for each of the commands that will be registered. The interface will be implemented internally by Scene Composer. The plugin can then register the commands using the UICommandManager service. This can be done either directly or using the helper class CommandDefinitionCollection that simplifies the registering of multiple commands.

The command can be invoked from code, (see other more complex examples for a demonstration), or can be displayed in the menu or used as a shortcut. This is done by attaching metadata to the command, as displayed in the example.

```
public void Initialize(IUnityContainer unityContainer)
{
  var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

  //Creates a collection of ui element definitions.
  var uiElementCollection = new UIDefinitionCollection()
  {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialogView, Cu
  };
  // We register the elements of the collection with the UIDefinitionManager
  uiElementCollection.RegisterAll(uiDefManager);

  // We create the command collection and add the command definitions.
  var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
  {
     {
        c => c.ContextCommand,
        new ManagedCommand<IBitmap>()
        {
           ExecuteHandler = o=>
           {
           },
           Metadata = new MetadataCollection()
           {
              new ContextMenuOption()
              {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
              },
           new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
           }
        }
     },
     {
        c=>c.ShowDialogCommand,
        new ManagedCommand()
        {
          ExecuteHandler = (o)=>
          {
```

```
            var dlgSrv = unityContainer.Resolve<IDialogSvc>();
            var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
            dialogVM.Message = "Custom Dialog";
            if (dlgSrv.ShowDialog(dialogVM) == true)
            {
              dlgSrv.ShowMessageBox(dialogVM.Message);
            }
          },
          Metadata = new MetadataCollection()
          {
            new MainMenuOption()
            {
              MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 350 },
            },
            new ToolTip("Invokes a custom dialog from a plugin"),
            new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | ModifierKeys.Shi
          }
        }
    }
  };
  cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
 }
```

The dialog window

This custom dialog will contain a button that will exacute a command to display a messagebox.

```
this.ShowMessageBox = new ManagedCommand()
{
  ExecuteHandler = o =>
  container.Resolve<IDialogSvc>().ShowMessageBox
  (
  "Message box from a plugin",
  "Message box title",
  MessageBoxButton.OK,
  MessageBoxImage.Exclamation
  )
 };
```

The full code for the example:

CustomUIConfigPlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using FTC.SDKInterface.Plugins;
using FTC.SDKInterface.Selections;
using FTC.SDKInterface.Services;
using Feat.UIComposition;
using Feat.UIComposition.Common;
```

```
using Feat.UIComposition.Config;
using Feat.UIComposition.Paneling;
using FTC.SDKInterface.Constants;
using UIPlugins.PluginWithMenuAndDialog.CustomDialog;
using Feat.UIComposition.Dialogs;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Menus;
using Feat.UIComposition.Shortcuts;
using System.Windows.Input;
using Feat.CAL.Services;
using Feat.UIComposition.ContextMenus;
using Feat.Core;
using FTC.SDKInterface.Solution;

namespace UIPlugins.PluginWithMenuAndDialog
{
   /// <summary>
   /// This is an example of a plugin that registers a command and displays a custom dialog ar
   /// </summary>
   public class PluginWithMenuAndDialogPlugin : IPluginService
   {
      public string Description
      {
         get { return "A plugin with a menu item and dialog"; }
      }
      public string Name
      {
         get { return this.GetType().Name; }
      }

      public string Version
      {
         get { return "1.1.1"; }
      }

      public IEnumerable<int> FeatureIds
      {
         get { return null; }
      }

      /// <summary>
      /// The initialization method has the role of registering with Scene composer different
      /// </summary>
      /// In this example the initliaze method will register a menu command and a custom dialo
      ///
      /// Registering Commands:
      /// To register a command first an interface that derives from ICommandContainer must be
      /// The interface will contain properties for each of the commands that will be register
      /// The interface will be implemented internally by Scene Composer.
      /// The plugin can then register the commands the UICommandManager service. This can be
      /// directly or using the helper class CommandDefinitionCollection that simplifies the r
      /// commands. \n
      ///
      /// The command can be invoked from code, (see other more complex examples for a demonst
```

```
/// or can be displayed in the menu or used as a shortcut. This is done by attaching met
/// as displayed in the example.
/// <param name="unityContainer"></param>
public void Initialize(IUnityContainer unityContainer)
{
   var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

   //Creates a collection of ui element definitions.
   // In this case we just register a dialog.
   // A ui element has 4 elements :
   // - a view model interface
   // - a view model class implementation
   // - a view interface
   // - a view class implementation
   var uiElementCollection = new UIDefinitionCollection()
   {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialogV
   };
   // We register the elements of the collection with the UIDefinitionManager
   uiElementCollection.RegisterAll(uiDefManager);

   // We create the command collection and add the command definitions.
   var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
   {
      {
         c => c.ContextCommand,
         new ManagedCommand<IBitmap>()
         {
            ExecuteHandler = o=>
            {
               //unityContainer.Resolve<FTC.SDKInterface.Selections.ISelectedScene>().Va
            },
            Metadata = new MetadataCollection()
            {
               new ContextMenuOption()
               {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
               },
               new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
            }

         }
      },
      {
         // We specify the command for which we are registering an implementation,
         // The type of c will be inferred to be IPluginCommands, so we don't have to sp
         c=>c.ShowDialogCommand,
         //
         new ManagedCommand()
         {
            ExecuteHandler = (o)=>
            {
               var dlgSrv = unityContainer.Resolve<IDialogSvc>();
```

```
                    // Invokes a dialog that was previously registered with the UIDefinitionM
                    // We first create the viewmodel for the dialog
                    var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
                    // We configure the view model properties
                    dialogVM.Message = "Custom Dialog";

                    // We show the dialog. The dialog window will be constructed based on the
                    // UIDefinitionManager service.
                    if (dlgSrv.ShowDialog(dialogVM) == true)
                    {
                       dlgSrv.ShowMessageBox(dialogVM.Message);
                    }
                 },
                 Metadata = new MetadataCollection()
                 {
                    // The main menu option to be created.
                    // The menu path is specifies the location where the menu should be displ
                    // The menu constant class contains the main menu definitions used by sce
                    // The plugin can an other main menu options too.
                    // The tokens of the path can be specified either with commas, or using m
                    // The each token of a path can be followed by an integer that represents
                    // in the menu. The option present in the scene composer menu, have order
                    // so plugins can add items in between the existing items
                    // A menu can be devided in several groups, in the example below, the ite
                    new MainMenuOption()
                    {
                       MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 35
                    },
                    // Specifies a tooltip for the command, used for the menu items and the k
                    new ToolTip("Invokes a custom dialog from a plugin"),
                    //Specifies a shortcut key for the command.
                    new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | Modifi
                 }
              }
           }
        };
        // We register the commands.
        cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
     }
  }

  /// <summary>
  /// The interface that contains the command definitions.
  /// </summary>
  /// The main role of this interface is to uniquely identify the command and to provide an e
  /// No implementation of this interface is required, Scene COmposer will generate a concret
  public interface IPluginCommands : ICommandContainer
  {
     ManagedCommand ShowDialogCommand { get; }
     ManagedCommand<IBitmap> ContextCommand { get; }
  }
}
```

ICustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The dialog view model interface. This interface can be used to exchange information bet
   /// </summary>
   public interface ICustomDialogViewModel
   {
      string Message { get; set; }

   }
}
```

ICustomDialogView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The interface for the dialog view. It contains no methods as the view does not communic
   /// </summary>
   public interface ICustomDialogView : IDialogView
   {
   }
}
```

CustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows;
using FTC.SDKInterface.Services;
using Feat.Core;
using System.Windows.Input;
using Feat.UIComposition.Commanding;
using Feat.CAL.Services;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
```

```
    class CustomDialogViewModel : ICustomDialogViewModel
    {
       public CustomDialogViewModel(IUnityContainer container)
       {
          // We create a simple command to demo using a Scene Composer message box
          // This is the way to show message box so that it will conform to the Scene Composer
          this.ShowMessageBox = new ManagedCommand()
          {
             ExecuteHandler = o =>
                container.Resolve<IDialogSvc>().ShowMessageBox
                (
                   "Message box from a plugin",
                   "Message box title",
                   MessageBoxButton.OK,
                   MessageBoxImage.Exclamation
                )
          };
       }

       /// <summary>
       /// The command for showing a dialog box, that will be bound to the view.
       /// </summary>
       public ICommand ShowMessageBox { get; set; }

       /// <summary>
       /// The message that will be retuned by the viewmodel.
       /// </summary>
       public string Message { get; set; }
    }
 }
```

CustomDialogView.xaml

```
 <!--
     The actual UI for the view. Most of the look and feel for the application is inherited by t
     The dialog window must however specify the Style property to ensure consistent look and fee
     The DialogWindow style resource is defined within Scene Composer
  -->

  <ui:DialogWindow
      x:Class="UIPlugins.PluginWithMenuAndDialog.CustomDialog.CustomDialogView"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:ui="clr-namespace:Feat.Core;assembly=Feat.Core"
      Title="Plugin dialog"
      Style="{DynamicResource DialogWindow}" >
     <StackPanel Margin="15">
        <Label Content="This is a custom dialog defined in a plugin"></Label>
        <StackPanel>
           <Label Content="Message:"></Label>
           <!--We use databiding to edit the MessageProperty-->
           <TextBox Text="{Binding Message}"></TextBox>
        </StackPanel>
```

```
        <!--We  use databinding to assign the ShowMessageBox command to the button-->
        <Button Command="{Binding ShowMessageBox}" Content="Show message Box" Margin="0,30,0,10"
        <StackPanel Orientation="Horizontal" Margin="10">
           <Button Click="OnOk" Margin="10" Content="OK"></Button>
           <Button Click="OnCancel" Margin="10" Content="Cancel"></Button>
        </StackPanel>
     </StackPanel>
  </ui:DialogWindow>
```

A plugins that defines a custom menu

Creating a plugin that displays a custom menu

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll, Feat.UIComposition.dll and Feat.Core.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 IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
 public class CustomMenuPlugin : IPluginService
 {

 }
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return "Custom Menu Definition"; }
}

public string Description
{
   get { return "A plugin that defines a custom menu"; }
}

public string Version
{
  get { return "1.0"; }
}
public void Initialize(IUnityContainer unityContainer)
{
    //register the menu config interface
    var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface
```

```
    ...
}
```

Config and commands interfaces

We will need to provide an interface derived from IConfiguration and an interface derived from ICommandContainer that defines the available commands.

```
public interface IMenuPluginConfig : IConfiguration
{
  ObservableCollection<string> Data { get; set; }
}

public interface ICustomPluginCommands : ICommandContainer
{
  ManagedCommand AddItem { get; }
  ManagedCommand RemoveItem { get; }
}
```

Then in the Initialize method:

Register the menu config interface

```
...
var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface<IMe
loader.ConfigLoaded += loader_ConfigLoaded;
...
```

Register the menu commands

```
 var cmdCollection = new CommandDefinitionCollection<ICustomPluginCommands>()
 {
  {
     c=>c.AddItem,
     new ManagedCommand()
     {
        ExecuteHandler = o=>
        {
           var data = loader.Instance.Data;
           data.Add("Item " + data.Count);
        },
        Metadata = new MetadataCollection()
        {
           new MainMenuOption()
           {
              MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Add" },
           }
         }
     }
  },
  {
```

```
      c=>c.RemoveItem,
      new ManagedCommand()
      {
         CanExecuteHandler = o=> loader.Instance.Data.Any(),
         ExecuteHandler = o=>
         {
            var data = loader.Instance.Data;
            data.RemoveAt(0);
         },
         Metadata = new MetadataCollection()
         {
            new MainMenuOption()
            {
               MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Remove" },
            },
         }
      }
   }
};
cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
```

And register a custom menu definition to hold the item list

```
  var menuDefMgr = unityContainer.Resolve<MenuDefinitionManager>();
  menuDefMgr.RegisterMenuDefinition(new CustomMenuDefinition()
  {
     MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "RegItems" },
  });

  loader.Instance.Data.CollectionChanged += Data_CollectionChanged;
```

The full code for the example:

CustomMenuPlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using FTC.SDKInterface.Plugins;
using System.Collections.ObjectModel;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Config;
using Feat.UIComposition;
using Feat.UIComposition.Menus;
using FTC.SDKInterface.Constants;

namespace UIPlugins.CustomMenu
{
  public class CustomMenuPlugin : IPluginService
  {
     public string Name
```

```
{
   get { return this.Description; }
}

public string Version
{
   get { return "1.0"; }
}

public string Description
{
   get { return "Custom Menu Definition"; }
}

public IEnumerable<int> FeatureIds
{
   get { return null; }
}

public FTC.SDKInterface.Services.IUnityContainer UnityContainer { get; set; }

public void Initialize(FTC.SDKInterface.Services.IUnityContainer unityContainer)
{
   var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterf
   loader.ConfigLoaded += loader_ConfigLoaded;
   this.UnityContainer = unityContainer;

   var cmdCollection = new CommandDefinitionCollection<ICustomPluginCommands>()
   {
      {
         c=>c.AddItem,
         new ManagedCommand()
         {
            ExecuteHandler = o=>
            {
               var data = loader.Instance.Data;
               data.Add("Item " + data.Count);
            },
            Metadata = new MetadataCollection()
            {
               new MainMenuOption()
               {
                  MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Add" },
               }
            }
         }
      },
      {
         c=>c.RemoveItem,
         new ManagedCommand()
         {
            CanExecuteHandler = o=> loader.Instance.Data.Any(),
            ExecuteHandler = o=>
            {
```

```
                    var data = loader.Instance.Data;
                    data.RemoveAt(0);
                },
                Metadata = new MetadataCollection()
                {
                    new MainMenuOption()
                    {
                        MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Remove" },
                    },
                }
            }
        }
    };
    cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());

    var menuDefMgr = unityContainer.Resolve<MenuDefinitionManager>();
    menuDefMgr.RegisterMenuDefinition(new CustomMenuDefinition()
    {
        MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "RegItems" },
    });

    loader.Instance.Data.CollectionChanged += Data_CollectionChanged;
  }

  void Data_CollectionChanged(object sender, System.Collections.Specialized.NotifyCollectionC
  {
    var cmdContainer = this.UnityContainer.Resolve<UICommandManager>().GetCommandContainer<I
    cmdContainer.RemoveItem.RaiseCanExecuteChanged();
  }

  void loader_ConfigLoaded(object sender, ConfigLoadedEventArgs<IMenuPluginConfig> e)
  {
    if (!e.ConfigLoadedSuccessfully)
    {
      e.Instance.Data = new ObservableCollection<string>();
    }
  }
}

public interface IMenuPluginConfig : IConfiguration
{
  ObservableCollection<string> Data { get; set; }
}

public interface ICustomPluginCommands : ICommandContainer
{
  ManagedCommand AddItem { get; }
  ManagedCommand RemoveItem { get; }
}
}
```

CustomMenuDefintion.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.UIComposition.Menus;
using System.Windows.Controls;
using Feat.UIComposition.Config;
using System.Collections.ObjectModel;
using Feat.UIComposition.Commanding;
using Feat.CAL.Services;
using Feat.Utils;
using System.Windows;

namespace UIPlugins.CustomMenu
{
  class CustomMenuDefinition : MenuDefinitionBase
  {
     protected override IMenuItemManager CreateMenuItemManger(MenuManager menuManger)
     {
        return new CustomMenuManager(this);
     }

     public override string UniqueName
     {
        get { return this.GetType().AssemblyQualifiedName + this.MenuPath; }
     }
  }

  class CustomMenuManager : IMenuItemManager
  {
     public ObservableCollection<string> items;
     private IDialogSvc dialogSvc;

     public CustomMenuDefinition Definition { get; set; }

     public CustomMenuManager(CustomMenuDefinition cmDef)
     {
        this.Definition = cmDef;
        this.dialogSvc = this.Definition.UnityContainer.Resolve<IDialogSvc>();
        this.items = this.Definition.UnityContainer.Resolve<IConfigurationManager>().
           GetConfiguration<IMenuPluginConfig>().Data;
        items.CollectionChanged += items_CollectionChanged;
     }

     void items_CollectionChanged(object sender, System.Collections.Specialized.NotifyCollectior
     {
        this.Invalidate();
     }

     MenuItem rootMenuItem;
     public void Attach(MenuManager menuMgr, ItemsControl parentMenuItem, object startingSibling
     {
        this.rootMenuItem = new MenuItem()
        {
```

```
            Header = "ItemList"
         };
         parentMenuItem.Items.Add(rootMenuItem);
         this.Invalidate();
      }

      public bool IsItemVisible
      {
         get
         {
            return this.items.Any();
         }
      }

      public event EventHandler HasItemsChanged;

      public void Invalidate()
      {
         this.rootMenuItem.Items.Clear();
         items.
            Select(s => new MenuItem()
            {
               Header = s,
               Command = new ManagedCommand()
               {
                  ExecuteHandler = o =>
                     this.dialogSvc.ShowMessageBox(s, s, MessageBoxButton.OK, MessageBoxImage.Inf
               }
            }).
            ForEach(i => this.rootMenuItem.Items.Add(i));

         if (this.HasItemsChanged != null)
         {
            this.HasItemsChanged(this, EventArgs.Empty);
         }
      }
   }
}
```

A simple plugin that consumes events

Creating a plugin that consumes events

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll and Feat.Core.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 IEventListener interface. This is the type that will be instantiated by Scene Composer.

```
public class SimpleEventsConsumer : IEventListener
{
}
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return "SC event consumer plugin"; }
}

public string Description
{
   get { return "A plugin that consumes SC events and shows a message box on each event"; }
}

public string Version
{
  get { return "1.1.0"; }
}
public void Initialize(IUnityContainer unityContainer)
{
   //resolve the IDialogSvc interface. We will use it to show simple message boxes.
  dlgSvc = unityContainer.Resolve<IDialogSvc>();
}
```

Specifying what events should be handled by the plugin.

We implement the EventsToSubscribe property of the IEventListener interface. This property specifies a set of events that Scene Composer should pass to the plugin.

```
public EventId EventsToSubscribe
{
  get { return EventId.AppShutdown | EventId.AssetGeneration | EventId.ImportFinished | EventId
}
```

Implementing the event handlers of IEventListener

Next we implement event handler methods.

```
public void OnAssetGeneration(string assetFile, bool isTargetAsset, OperationStatus status)
{
  ShowMessage(String.Format("Asset generation for '{0}'  {1}.", assetFile, status));
}

public void OnImportFinished()
{
  ShowMessage("Import finished.");
}

public void OnShutdown()
{
  ShowMessage("App shutting down.");
}

public void OnSolutionChanged(FTC.SDKInterface.Solution.ISolution solution)
{
  ShowMessage("Solution changed!");
}
```

The ShowMessage method uses IDialogSvc to show a messagebox.

```
private void ShowMessage (string message)
{
  Application.Current.Dispatcher.BeginInvoke(new Action( () =>
  {
    dlgSvc.ShowMessageBox(message, "EventsPlugin");
  }), null );
}
```

The full code for the example:

SimpleEventsConsumer.cs

```
using Feat.CAL.Services;
using FTC.SDKInterface.Plugins;
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace SCEventsPlugin
{
  public class SimpleEventsConsumer : IEventListener
  {
     private IDialogSvc dlgSvc;

     private void ShowMessage(string message)
     {
        Application.Current.Dispatcher.BeginInvoke(new Action(() =>
        {
           dlgSvc.ShowMessageBox(message, "EventsPlugin");
        }), null);
     }

     public EventId EventsToSubscribe
     {
        get { return EventId.AppShutdown | EventId.AssetGeneration | EventId.ImportFinished | E
     }

     public void OnAssetGeneration(string assetFile, bool isTargetAsset, OperationStatus status
     {
        ShowMessage(String.Format("Asset generation for '{0}'  {1}.", assetFile, status));
     }

     public void OnImportFinished()
     {
        ShowMessage("Import finished.");
     }

     public void OnShutdown()
     {
        ShowMessage("App shutting down.");
     }

     public void OnSolutionChanged(FTC.SDKInterface.Solution.ISolution solution)
     {
        ShowMessage("Solution changed!");
     }

     public string Description
     {
        get { return "A plugin that consumes SC events and shows a message box on each event";
     }

     public IEnumerable<int> FeatureIds
     {
        get { return new int[0]; }
     }
```

```
      public void Initialize(FTC.SDKInterface.Services.IUnityContainer unityContainer)
      {
         //resolve the IDialogSvc interface. We will use it to show a message boxes.
         dlgSvc = unityContainer.Resolve<IDialogSvc>();
      }

      public string Name
      {
         get { return "SC event consumer plugin"; }
      }

      public string Version
      {
         get { return "1.1.0"; }
      }
   }
}
```

Custom Properties Support

Custom Properties Support

The plugins can register (extra) properties for solution items. These properties can be edited using the property grid and are stored in the solution files.

The Type Manager service contains a method to register properties and another one to retrieve all properties registered (by the plugins) for a type. A reference to this service can be obtained by the plugin in the initialization method using the Unity Container.

DepProperty RegisterProperty(Type ownerType, string name, Type type)

Registers a new property for the specified type.

- ownerType is the type for which the property will be registered. If ownerType is an interface the property will be available for all classes which implement that interface. If ownerType is a class the property will be available for all classes which are derived from it. For example if ownerType is typeof(IScene) the property will be available for all scenes and composites in the solution.
- name specifies the name of the property.
- type specifies the type of the property. For example typeof(string) or typeof(float). The object returned by this method should be stored by the plugin because it is required to get and set the value of the property.

```
    public DepProperty ReqIdProperty;
    public void Initialize(IUnityContainer unityContainer)
    {
       var typeManager = unityContainer.Resolve<ITypeManagerSvc>();
       this.ReqIdProperty = typeManager.RegisterProperty(typeof(IScene), "ReqId", typeof(strir
    }

    public void OnSolutionChanged(SceneComposer.SDKInterface.Solution.ISolution solution)
    {
       foreach (var scene in solution.Scenes)
       {
          //get property value
          var reqId = scene.GetValue(this.ReqIdProperty) as string;

          //set property value
          scene.SetValue(this.ReqIdProperty, "REQ_ID_");

          //reset property value
          scene.UnsetValue(this.ReqIdProperty);
       }
```

```
    }
```

IEnumerable<DepProperty> GetRegisteredProperties(Type ownerType)

Returns an enumerable with all the custom properties registered (by plugins) for a specific type including those registered by other plugins and those registered for subtypes of ownerType.

FTC Command (Optional)

Command Line Automation with FTC Command

FTCCmd is a command line tool that supports operations like import and the generation of assets without the need to open SceneComposer GUI.

It is delivered as an ***optional add on*** to CGI Studio with the following content:

- *<cgi-studio-root>/bin/SceneComposer/FTCCmd.exe*
- *<cgi-studio-root>/bin/SceneComposer/FTCCmd.exe.config*

FTCCmd can be launched by using the Windows Command Prompt.

> FTCCmd.exe relies on standard SceneComposer binaries, hence it is only functional when located in the same folder as SceneComposer.exe and related dynamic link libraries.

FTCCmd Arguments

The FTCCmd syntax basically supports following command line arguments:

```
- FTCCmd /? - displays generic info about the possible modes (Import, Info
, Generate, ExportScl)
- FTCCmd {Mode} /? - displays detailed info about the specified mode
- FTCCmd {Mode} {SolutionFile} {Options} - performs an operation on the given solution as specif
```

Note the following syntax rules:

- {} = Any content which is included between curly brackets is *mandator*
- [] = Any content which is included between square brackets is *optional*
- [...] = The suspension point indicates that the content can be *repeated*
- For any path the "" (quotation marks) have to be used.

```
C:\SceneComposer>FTCCmd /?
FTCCmd /?
 --or--
```

```
FTCCmd {Mode} /?
 --or--
FTCCmd /v
 --or--
FTCCmd /version
 --or--
FTCCmd {Mode} {SolutionFile} {Options}

  Mode                Specifies the mode (Info, Import, Generate, ExportScl).
  SolutionFile        The solution file which will be processed.
  Options             See help specific for each mode.
  /SH,/SCHost         Specifies the path to the SCHost.dll file that will be taken into
account.

C:\SceneComposer>
```

There are four operation modes supported:

- Info
- Import
- Export SCL
- Generate

Arguments related to FTCCmd configuration and license handling

- /Config
- /Licenses

Get Info, Import or Generate asset for old solutions using Convert option

- /Convert

Get a piece of information about version both in FTCLauncher and FTCCmd

- /Version

Specify full file path to SCHost.dll file for Scene Composer

- /SCHost

FTCCmd Specifying which SCHost.dll : /SH,/SCHost

This command allows you to specify the full file path to the SCHost.dll file to be used by FTCCmd.

Example:

FTCCmd {Mode} {SolutionFile} {Options} /SCHost {SCHost.dll Path}

```
FTCCmd ExportScl d:\Libraries\Materials.scs /outscl d:\ExportedLibraries\Materials.scl /SCHost '
```

FTCCmd Operation Mode: Info

Use the *Info* operation mode of FTCCmd to display some basic information about the content of a given solution (for example: the scene contained in the accessed solution, the used animations, the widgets which are used in the given solution and their properties).

```
C:\SceneComposer>FTCCmd info
Displays a list with scenes and animations contained in a solution.

FTCCmd Info {SolutionFile} [/Convert Yes|No]

  SolutionFile        The solution file which will be processed.
  /Convert            Converts the solution if necessary (Yes, No, default No).

C:\SceneComposer>
```

Example:

```
FTCCmd Info "C:\Documents and Settings\User\Desktop\Solution\Solution.scs"
```



FTCCmd Operation Mode: Import

Use the *Import* operation mode of FTCCmd to import content like FBX files or images to a given solution. Following options are supported:

```
C:\SceneComposer>FTCCmd Import
Imports content into a solution.

FTCCmd Import {SolutionFile} /Importer Fbx|Image /Input {File1} [{File2} ...] /Destination
{DestinationFolder} [/Flags {Flag1} [{Flag2} ...]] [/Convert Yes|No] [/Config {ConfigFile}]

  SolutionFile        The solution file which will be processed.
  /Importer           Specifies the importer to use (Fbx, Image).
  /I,/Input           Specifies the list of files to process. The following wildcards are
supported: '*' - zero or more characters, '?' - zero or one characters. Example: *.jpg
  /B,/BitmapProfile   Specifies the bitmap profile to be applied to the imported images (full
name). Example: "/Global/Config/Profiles/BitmapProfiles/BitmapProfile"
  /D,/Destination     Specifies the destination folder from the solution where the imported
content will be put.
  /F,/Flags           Specifies the import flags (ClearDestination,
PreserveLibraryItemAttributes, ShowInImports, UnlockItems).
```

```
  /Convert            Converts the solution if necessary (Yes, No, default No).
  /Cfg,/Config        Specifies the configuration file.

Example:
FTCCmd Import d:\Solutions\SolutionAuto\SolutionAuto.scs /Importer Fbx /Input
d:\Solutions\_importable\CarGeneric\CarGeneric.FBX /Destination /MyModule/Imports/CarGeneric

C:\SceneComposer>
```

Optional Import flags:

- *ClearDestination*: Deletes existing items in the destination folder
- *PreserveLibraryItemAttributes*: Preserves item attributes
- *ShowInImports*: Makes the imported objects visible in Imports panel

Wildcard Support:

If there are many files which must be imported we can use the available wildcards - characters that can be used as a substitute for an entire class of characters. For instance, if in a folder we have many graphic files named "img1.png", "img2.png", "img3.png" and so forth, and if it is necessary to import all of them we can do this by specifying through the wildcard (*) that we need to import all the files which respect the rule "img*.png".

Available wildcards:

- *(*) Star wildcard*: used for 0 or many characters (any)
- *(?) Question mark wildcard*: used for 0 or exactly one character (any)

If it is necessary to set the bitmap attributes, following option is available:

```
FTCCmd ... /BitmapProfile <bitmap_profile_full_name>
```

Examples:

```
FTCCmd Import "D:\Work\Solutions\SolutionImportBitmap\SolutionImportBitmap.scs" /Importer image
FTCCmd Import "D:\Work\Images\rust.png" /BitmapProfile "/Global/Config/Profiles/BitmapProfiles/E
/Destination "/MyModule/Imports/Import5"
```

FTCCmd Operation Mode: Export SCL

This mode will allow you to export a specific solution to an .scl file. The specified output solution must be of type SCL in order to succeed.

```
C:\SceneComposer>FTCCmd ExportSCl
Exports the specified solution to an .scl file. The specified solution must be of type SCL in
order to succeed.

FTCCmd ExportScl {SolutionFile} /outscl {OutputFilePath}

  SolutionFile               The solution file which will be processed.
  /outscl OutputSclFilePath  Specifies the file path where the SCL is to be generated
  /Convert                   Converts the solution if necessary (Yes, No, default No).

Example:
FTCCmd ExportScl d:\Libraries\Materials.scs /outscl d:\ExportedLibraries\Materials.scl

C:\SceneComposer>
```

- SolutionFile: The file path to the solution file (.scs) to be exported.
- OutputFilePath: The file path including the (.scl) file name where the SCL is to be generated.

Example:

```
FTCCmd ExportScl d:\Libraries\Materials.scs /outscl d:\ExportedLibraries\Materials.scl /SCHost '
```

FTCCmd Operation Mode: Generate

Use the *Generate* operation mode of FTCCmd to generate a simulation or target asset from a given solution. Following options are supported:

```
C:\SceneComposer>FTCCmd Generate
Generates an asset file from the solution.

FTCCmd Generate {SolutionFile} /ShaderCompiler Target|Simulation /Asset {AssetFile}
[/AssetProfile {AssetProfileFullName}] [/CustomId {Id}] [/FloatFormat {Format}] [/Convert
```

```
Yes|No] [/Config {ConfigFile}] [/Licenses None | {Lic1} [{Lic2} ...]]

  SolutionFile        The solution file which will be processed.
  /SC,/ShaderCompiler Specifies the shader compiler to use (Simulation, Target).
  /FF,/FloatFormat    Specifies the float number format to use.
  /AF,/Asset          Specifies the asset file which is going to be generated.
  /AP,/AssetProfile   Specifies the full name of the asset profile (from solution) to be
activated
  /Id,/CustomId       Specifies the custom id written to the asset header (long integer,
default 0).
  /Convert            Converts the solution if necessary (Yes, No, default No).
  /Cfg,/Config        Specifies the configuration file.
  /Lic,/Licenses      Specifies the licenses which will be acquired (2D, 3D, Globalization,
SmartImporter, AssetLibraryVerification, Safety). If this parameter is not specified and no
license configuration file is found (see /Config) all licenses will be acquired.

Example:
FTCCmd Generate d:\Solutions\SolutionAuto\SolutionAuto.scs /ShaderCompiler Target /Asset
d:\AssetTarget.bin

C:\SceneComposer>
```

Optional Generate flags:

- *CustomId*: Asset verification value written in an asset header section
- *FloatFormat*: Information about the representation of float numbers in the asset; all the formats are platform dependent

Sample Asset Generation: Using /ShaderCompiler /Asset /AssetProfile parameters

Example:

```
FTCCmd Generate "C:\sample\2D Basic.scs" /ShaderCompiler Simulation /Asset Asset.bin /AssetProfi
```

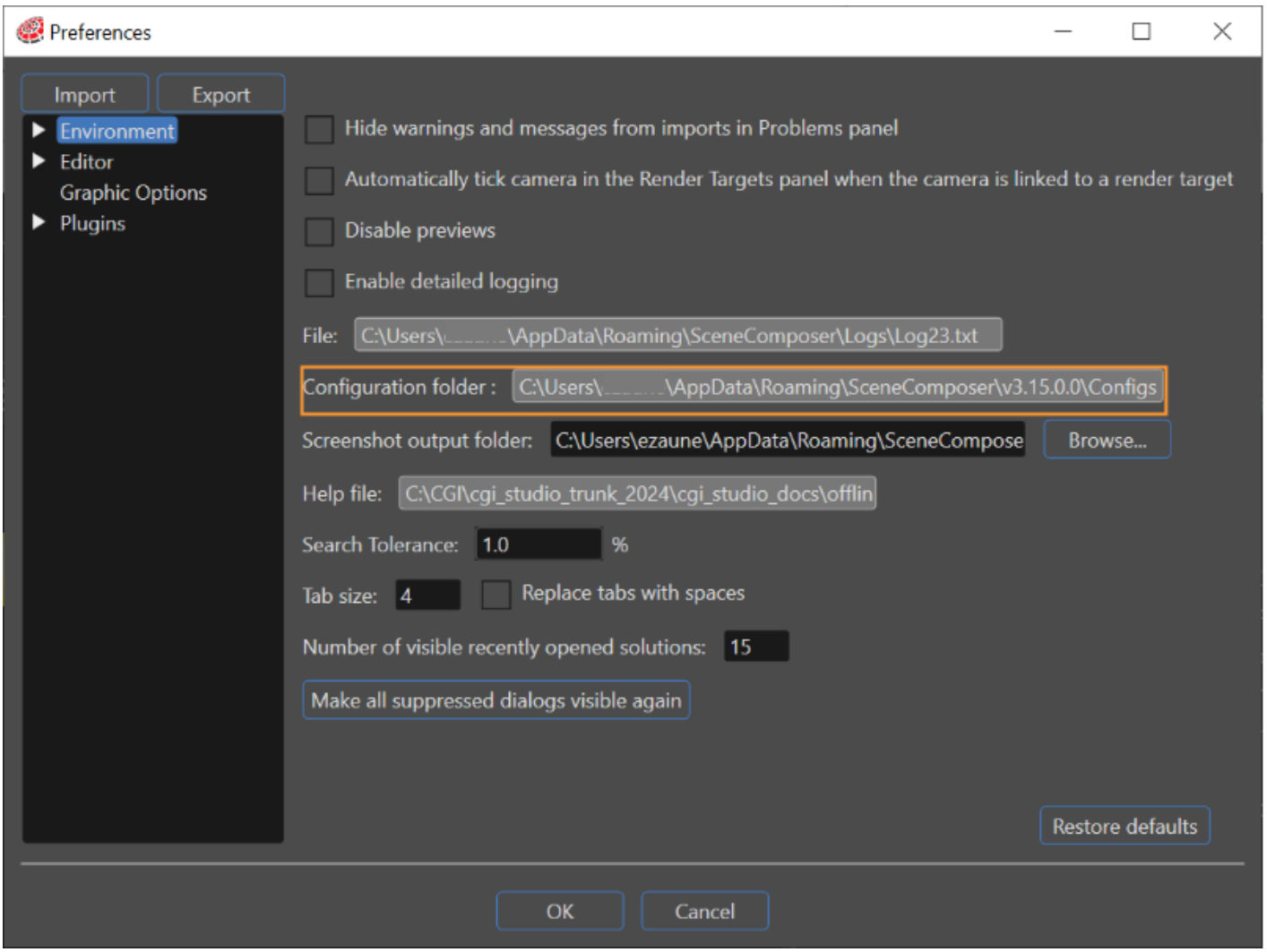
- /SC,/ShaderCompiler - Specifies the shader compiler to use (Simulation, Target).
- /AF,/Asset - Specifies the asset file name that is going to be generated.

- /AP,/AssetProfile - Specifies the full name of the asset profile from solution (.scml) to be activated.

FTCCmd Configuration: /Config

To reuse all the user specific persistent SceneComposer settings in an FTCCmd session, use the */Config* argument to specify the path to the SceneComposer configuration folder.

The configuration folder valid for a given SceneComposer GUI session can be found in "File" > "Preferences" section "Environment":



Instead of the full configuration folder, an exported .sccfg file can also be used as a /Config argument value. Use the "Export all" button to create a .sccfg file.

Examples:

```
FTCCmd ... /Config C:\Documents and Settings\...\...
FTCCmd ... /Config FTCSettings.sccfg
```

FTCCmd License Handling: /Licenses

By default - if neither /Config nor /Licenses arguments are given in an FTCCmd call - FTCCmd will acquire all licenses before the desired operation is executed.

- Refer to Using Licenses to learn about available licenses.

To adapt this behaviour, following options are available:

```
FTCCmd ... /Config {path to config folder or .sccfg file}
```

In this case, the license settings from the SceneComposer GUI can be applied to FTCCmd.

```
FTCCmd ... /Licenses 2D 3D Globalization
```

In this case, only the specified licenses ("2D", "3D", "Globalization", "PhotoshopImporter", "AssetLibraryVerification") will be acquired by FTCCmd.

```
FTCCmd ... /Licenses None
```

In this case, no license at all will be acquired by FTCCmd.

FTCCmd Convert solution: /Convert

By default FTCCmd will not convert old solutions. In order to get information, import or generate asset for old solutions, the following option is available:

```
FTCCmd .../Convert Yes
```

FTCCmd Version solution: /Version

A command which gives a piece of information about version both in FTCLauncher and FTCCmd:

```
FTCCmd .../Version
```

Automation API

Automation API

SceneComposer provides an interoperable interface, accessible from any .NET language binding that support a progressive and evolving level of automation.

SCManager exposes the automation API enabling an application developer to write an own program which can load SceneComposer solutions and generate asset files. In order to use the operations provided by this API, refer to following dlls:

- SceneComposer.Automation.dll
- SceneComposer.SDKInterface.dll

The API offers a singleton instance of SCManager which allows to instantiate a single object of class SCManager and use the methods provided.

Useing Automation API:

- Create a C# application
- Add SceneComposer.Automation.dll and SceneComposer.SDKInterface.dll as references in the C# project.
- Reference the dlls in the C# application:
  - using SceneComposer.SDKInterface.Solution;
  - using SceneComposer.Automation;

SceneComposer.Automation.dll

SceneComposer.Automation.dll provides the following methods and properties:

- SCManager.Instance.Initialize(): Performs the initialization of all services.
- SCManager.Instance.Solution: Returns the current solution or null if no solution is loaded.
- SCManager.Instance.LoadSolution(solutionFileName): Loads and returns a new solution from the specified solution file, which can further be accessed and managed through the query interface.
- SCManager.Instance.GenerateAsset(assetFileName, solution.Scenes): Generates an asset file based on the current solution. The asset file will contain all items which have the

"Always Export" flag set or are referred by other included items. In addition the asset file will contain the specified extra scenes with all their dependencies. GenerateAsset(assetFileName, scenesList) requests the path of the asset as a parameter. Its function is to generate the asset file at the specified location on the computer. The second parameter represents a list of the extra scenes that you want to include.

SceneComposer.SDKInterface.dll

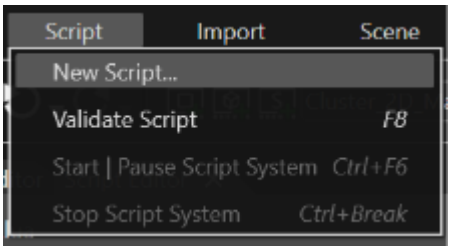
SceneComposer.SDKInterface.dll provides the methods that allow to manage and query the solution received with the LoadSolution() method or from the Solution property.

The query interface provides the following information:

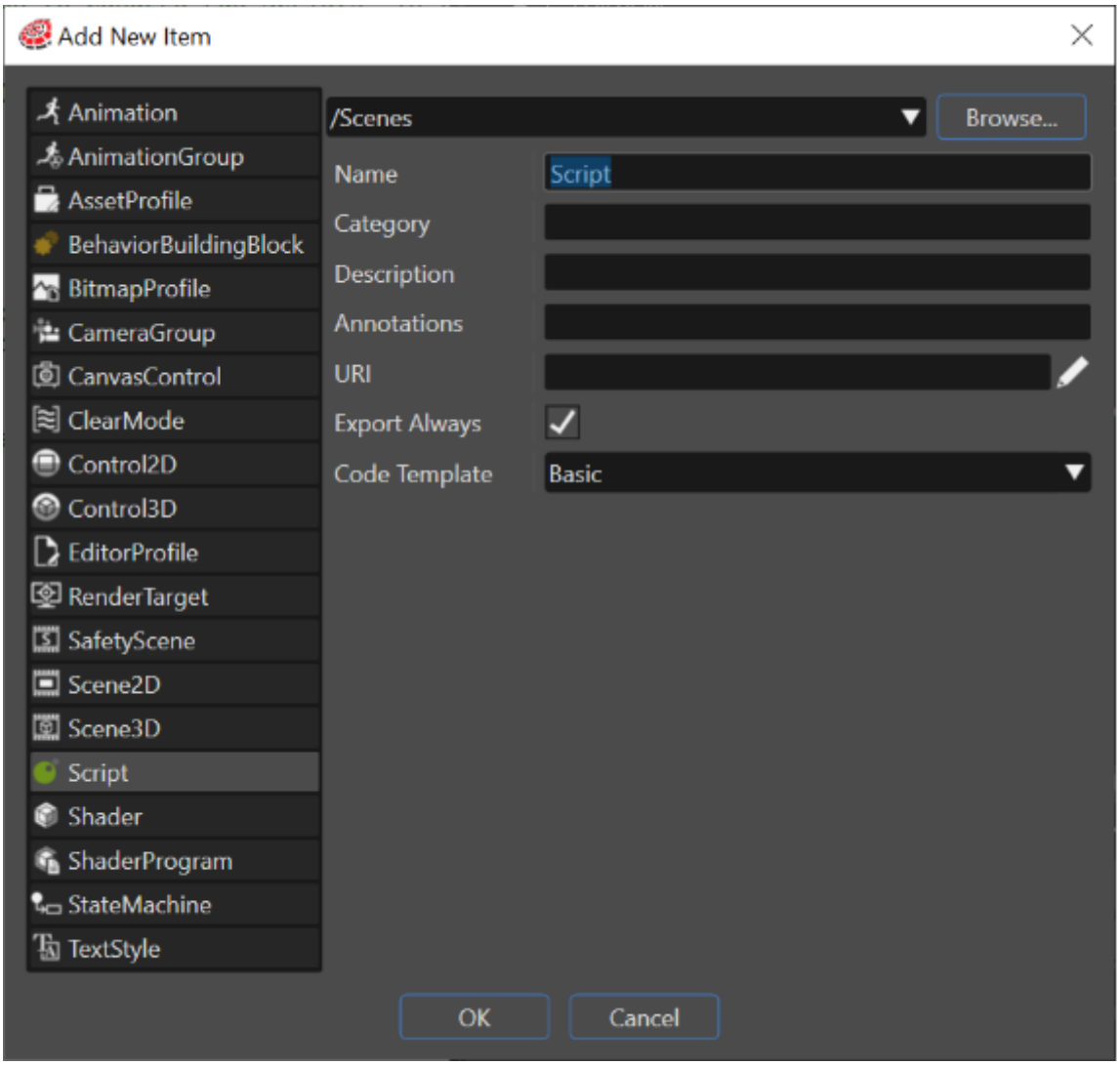
- List of scenes / Scene by name iterating on elements of the list
- List of animations / Animation by name

Scripting

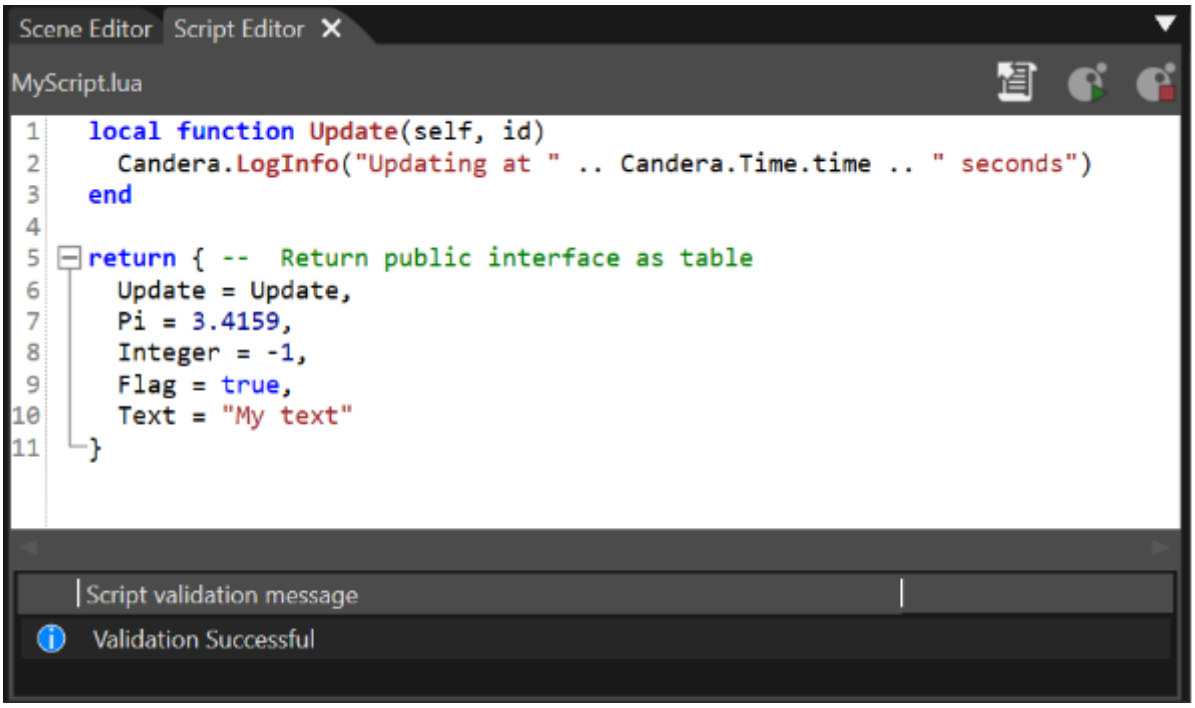
An empty script can be created via "New Script" menu item or the toolbar.



Alternatively, scripts can be added to a solution via "Add" > "New item" option by using the Scene Explorer context menu.

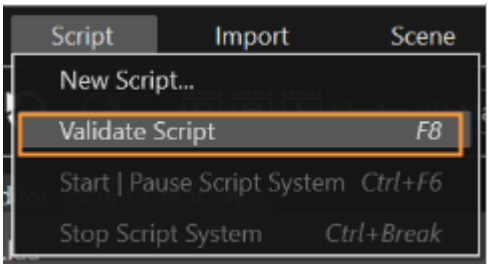


SceneComposer offers a script editor - which is loaded when double clicking on a "Script Component" - similar to shader editor.



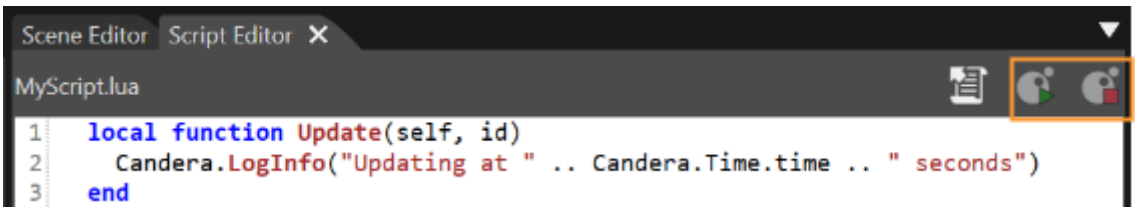
Scripts, written in the LUA programming language, can be "validated" by pressing F8 or by selecting Validate Script from the Script menu.

> "Script Editor" is not cleared if the script loaded inside is deleted from solution.



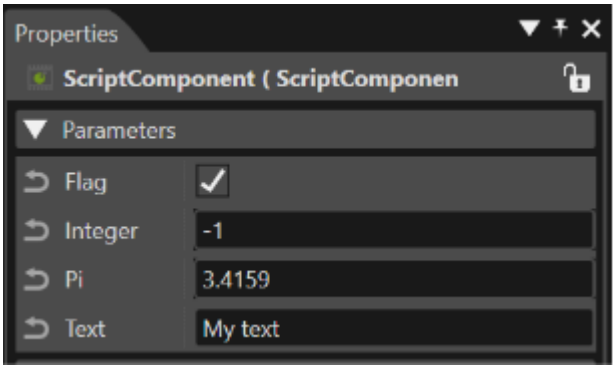
After an unsuccessful validation an error is displayed in the output and the LUA Validator message is shown in the Problems panel. After a successful validation no errors will be shown.

Any validated script can be activated by using the control interface (Play, Pause, Resume, Stop) available in the toolbar.

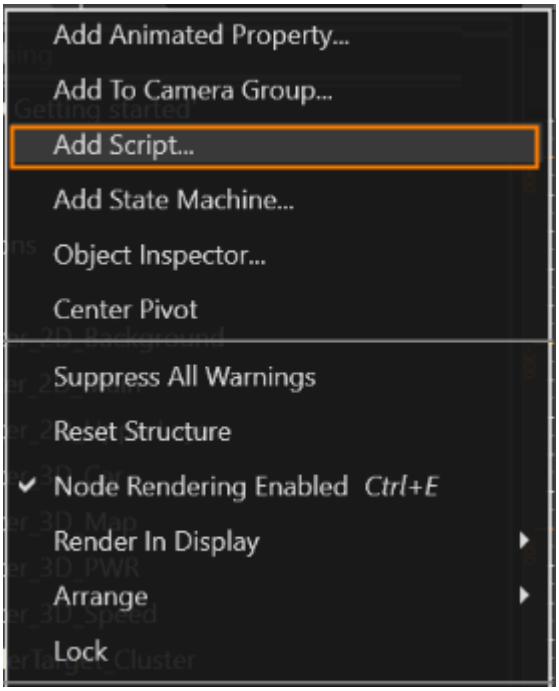


Dragging a script on a node will add a Script Component to that node (below Appearances). If the Script is successfully validated, the Script Component will expose the script's parameters in the

Property Grid.



If - at least - one valid script exists, it is possible to select from the context menu of a node the option "Add Script...". This it would open a "Choose Item" dialog and validate the selection of a "Script" object.



> Locking a "Script" item will not also lock "Script Editor".

Easy Mode Generator

> This page used to describe old Easy Mode, which used the "SceneComposerConfigurator.exe" file to generate configuration files (CGI Studio 3.12 and earlier). With the improvement, Easy Mode feature has been implemented in Preferences for easier configuration. For details on the newly implemented Easy Mode, please refer to Easy Mode.