

Adding plugins

Next Generation | Version: November 20, 2015 | Example: Available upon request

Support
support@visualcomponents.com

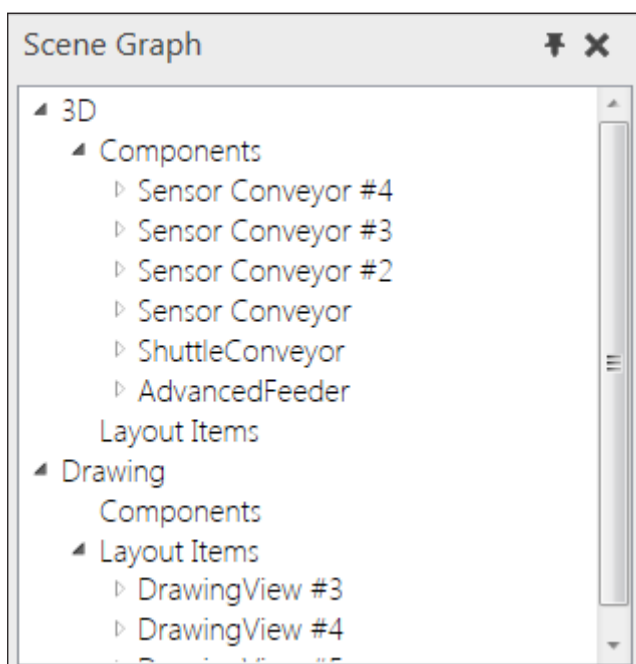
Community
community.visualcomponents.net

Essentials supports the development of system components which can be deployed as plugins that add functionality.

At its core, an Essentials plugin implements and is exported as a type of **IPlugin** interface. The plugin can operate in the background or be deployed with a user interface during runtime. In terms of object manipulation, the data of a simulation can be manipulated and stored in collections, which can be updated using event handlers tied to methods. Components and other objects native to Essentials can be created entirely using API.

The topics covered in this tutorial include:

- Adding new functionality and plugins that operate in the background during runtime.
- Developing plugins which can be attached to the main window of Essentials.
- Manipulating and storing data related to components and layouts loaded in a simulation.
- Creating components and other objects using Essentials API.
- Developing a custom scene graph panel for a simulated environment.



Creating a project in Visual Studio

Visual Studio 2013 and Visual C# are used to create a plugin that displays the data structure of simulated environments.

1. Run **Visual Studio** as an **administrator**.
2. Create a new **Class Library** project which uses .NET Framework 4.5 or higher and is named **SceneGraphPanel**.
3. Access the properties of your project, and then do the following:
 - In the **Application** tab, add a **UX.** prefix to your assembly name. This will id your assembly as an extension of the VisualComponents.UX namespace.

Assembly name:	Default namespace:
UX.SceneGraphPanel	SceneGraphPanel
Target framework:	Output type:
.NET Framework 4.5	Class Library

- In the **Build** tab, under **Output**, set **Output path** to be the directory of your Essentials program files.
 - In the **Debug** tab, under **Start Action**, set **Start external program** to be the executable file of Essentials.
4. Add references to the following assemblies and namespaces:
 - **Create3D.Shared** that is available in your Essentials program files.
 - **System.ComponentModel.Composition** that is available in the .NET Framework.
 - **UX.Shared** that is available in your Essentials program files.
 5. (Optional) Delete the **Class1.cs** from your project.

Developing a simple plugin

It is important to understand that an Essentials plugin does not require a user interface. Any client object (plugin) that implements and is exported as a type of `IPlugin` interface can be automatically imported by Essentials. To demonstrate this concept, complete the following steps.

1. Add a new **Class**, and name it **HelloWorld**.
2. In **HelloWorld**, type the following code to define a simple plugin that prints "Hello World" to the Output panel of Essentials.

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

namespace SceneGraphPanel
{
    using System.ComponentModel.Composition;
    using VisualComponents.Create3D;
    using VisualComponents.UX.Shared;

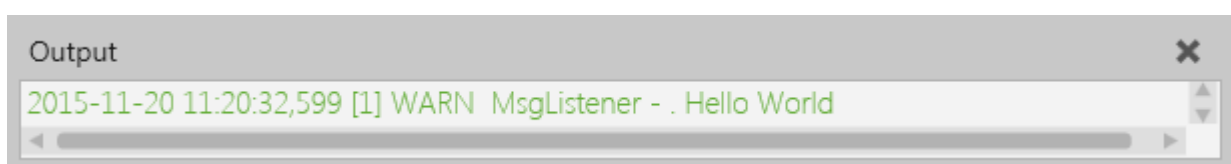
    [Export(typeof(IPlugin))]
    class HelloWorld: IPlugin
    {
        [Import]
        private Lazy<IApplication> _application = null;

        public void Exit()
        {
        }

        public void Initialize()
        {
            this._application.Value.WriteLine("Hello World");
        }
    }
}
```

NOTE! You can use the `Initialize()` and `Exit()` methods to receive notifications about the application's lifecycle.

3. Start debugging your project, and then verify the Hello World message is printed in the Output panel of Essentials, and then stop debugging your project.



4. (Optional) Delete the **HelloWorld** class from your project.

Collecting simulation data

The hierarchy of the 3D world, drawing world and components are tree structures. Therefore, you can collect and sort this data in collections as well as visualize the data and its hierarchy in TreeView controls.

1. In your project, add a reference to **Caliburn.Micro** which is available in your Essentials program files.
2. Add a new **Class**, and name it **TreeNodeViewModel**.
3. In **TreeNodeViewModel**, add using statements for **Caliburn.Micro**, **System.Collections.Generic**, **System.Collections.ObjectModel**, and **VisualComponents.Create3D**.
4. Set **TreeNodeViewModel** to inherit the **PropertyChangedBase** class available in the Caliburn.Micro assembly. This will help you implement functionality for updating your TreeView control.
5. Type the following code to complete the **TreeNodeView** class which is declared as public.

```
public class TreeNodeViewModel : PropertyChangedBase
{
    protected string _header;
    public string Header
    {
        get { return _header; }
        set
        {
            if (value != _header)
            {
                _header = value;
                NotifyOfPropertyChanged(() => Header);
            }
        }
    }

    protected ObservableCollection<TreeNodeViewModel> _children;
    public ObservableCollection<TreeNodeViewModel> Children
    {
        get { return _children; }
    }

    public TreeNodeViewModel()
    {
        _children = new ObservableCollection<TreeNodeViewModel>();
    }

    public TreeNodeViewModel(string name)
        : this()
    {
        _header = name;
    }
}
```

The remainder of the [Visualizing node hierarchy](#) section is mostly about implementing a set of collections to be bound to a TreeView control. Therefore, instructions and explanations are kept to a minimum; study the provided code as needed.

Property Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling the properties of components.

```
public class PropertyNodeViewModel : TreeNodeViewModel
{
    private IProperty _property;

    public PropertyNodeViewModel(IProperty property)
    {
        _property = property;
        string value = property.Value != null ? property.Value.ToString() : "NULL";
        _header = String.Format("{0} = {1}", property.Name, value);
    }
}
```

Feature Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling the features of components.

```
public class FeatureTreeNodeViewModel : TreeNodeViewModel
{
    private IFeature _feature;

    public FeatureTreeNodeViewModel(IFeature feature)
    {
        _feature = feature;
        _header = feature.Name;

        foreach (IFeature child in feature.Children)
        {
            _children.Add(new FeatureTreeNodeViewModel(child));
        }

        TreeNodeViewModel propertiesGroup = new TreeNodeViewModel("Properties");
        foreach (IProperty property in feature.Properties.Where(p => p.IsVisible))
        {
            propertiesGroup.Children.Add(new PropertyNodeViewModel(property));
        }
        _children.Add(propertiesGroup);
    }
}
```

Behavior Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling the behaviors of components.

```
public class BehaviorTreeNodeViewModel : TreeNodeViewModel
{
    private IBehavior _behavior;

    public IBehavior Behavior
    {
        get { return _behavior; }
    }

    public BehaviorTreeNodeViewModel(IBehavior behavior)
    {
        _header = behavior.Name;
        _behavior = behavior;

        TreeNodeViewModel propertiesGroup = new TreeNodeViewModel("Properties");
        foreach (IProperty property in _behavior.Properties.Where(p => p.IsVisible))
        {
            propertiesGroup.Children.Add(new PropertyNodeViewModel(property));
        }

        this.Children.Add(propertiesGroup);
    }
}
```

Layout Item Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling layout items.

```
public class LayoutItemTreeNodeViewModel : TreeNodeViewModel
{
    private ILayoutItem _layoutItem;
    public ILayoutItem LayoutItem
    {
        get { return _layoutItem; }
    }

    public LayoutItemTreeNodeViewModel(ILayoutItem layoutItem)
    {
        _layoutItem = layoutItem;
        _header = layoutItem.Name;

        TreeNodeViewModel propertiesGroup = new TreeNodeViewModel("Properties");
        foreach (IProperty property in _layoutItem.Properties.Where(p => p.IsVisible))
        {
            propertiesGroup.Children.Add(new PropertyNodeViewModel(property));
        }

        this.Children.Add(propertiesGroup);
    }
}
```

Component and Node Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling components.

```
public class SimTreeNodeViewModel : TreeNodeViewModel
{
    private ISimNode _node;

    public ISimNode Node
    {
        get { return _node; }
    }

    public SimTreeNodeViewModel(ISimNode simNode)
    {
        _node = simNode;
        _header = simNode.Name;

        if (simNode.IsComponentRoot)
        {
            ISimComponent component = simNode.Component;
            TreeNodeViewModel propertiesGroup = new TreeNodeViewModel("Properties");
            foreach (IProperty property in component.Properties.Where(p => p.IsVisible))
            {
                propertiesGroup.Children.Add(new PropertyNodeViewModel(property));
            }
            _children.Add(propertiesGroup);
        }

        TreeNodeViewModel nodeGroup = new TreeNodeViewModel("Nodes");

        foreach (ISimNode child in simNode.Children)
        {
            if (!child.IsComponentRoot)
            {
                nodeGroup.Children.Add(new SimTreeNodeViewModel(child));
            }
        }

        if (nodeGroup.Children.Count == 0)
        {
            nodeGroup.Children.Add(new TreeNodeViewModel("<empty>"));
        }
        _children.Add(nodeGroup);

        TreeNodeViewModel featureGroup = new TreeNodeViewModel("Features");
        featureGroup.Children.Add(new FeatureTreeNodeViewModel(simNode.RootFeature));

        if (featureGroup.Children.Count == 0)
        {
            featureGroup.Children.Add(new TreeNodeViewModel("<empty>"));
        }
    }
}
```

```
_children.Add(featureGroup);

TreeNodeViewModel behaviorGroup = new TreeNodeViewModel("Behaviors");
foreach (IBehavior behavior in _node.Behaviors)
{
    behaviorGroup.Children.Add(new BehaviorTreeNodeViewModel(behavior));
}

if (behaviorGroup.Children.Count == 0)
{
    behaviorGroup.Children.Add(new TreeNodeViewModel("<empty>"));
}
_children.Add(behaviorGroup);

}
}
```

World Node Objects

In the Code view for `TreeNodeViewModel`, add the following code to define a class for handling components and layout items in the 3D world and drawing world.

```
public class WorldTreeNodeViewModel :TreeNodeViewModel
{
    private ISimWorld _world;

    private TreeNodeViewModel _componentGroup;
    public TreeNodeViewModel ComponentGroup
    {
        get { return _componentGroup; }
    }

    private TreeNodeViewModel _layoutItemGroup;
    public TreeNodeViewModel LayoutItemGroup
    {
        get { return _layoutItemGroup; }
    }

    public WorldTreeNodeViewModel(ISimWorld world, string name)
    {
        _world = world;
        _header = name;

        _componentGroup = new TreeNodeViewModel("Components");
        this.Children.Add(_componentGroup);
        foreach (ISimComponent component in _world.Components)
        {
            _componentGroup.Children.Add(new SimTreeNodeViewModel(component.RootNode));
        }

        _layoutItemGroup = new TreeNodeViewModel("Layout Items");
        this.Children.Add(_layoutItemGroup);
        foreach (ILayoutItem layoutItem in _world.LayoutItems)
        {
            _layoutItemGroup.Children.Add(new TreeNodeViewModel(layoutItem.Name));
        }
    }
}
```

Creating ViewModel

A plugin can be designed to have a user interface and code-behind logic that follows the MVVM design pattern. In this case, a ViewModel-View pair can be created to define a dockable, scene graph panel for Essentials.

1. In your project, add a new **Class** item, and name it **SceneGraphPanelViewModel**. The "ViewModel" suffix is a way to identify the class as the logic for a user control with the same name that has a suffix of "View", for example SceneGraphPanelView.
2. In **SceneGraphPanelViewModel**, add the following code to define the plugin as a dockable panel and initialize the World tree node collection..

```
using Caliburn.Micro;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using VisualComponents.Create3D;
using VisualComponents.UX.Shared;

namespace SceneGraphPanel
{
    [Export(typeof(IPlugin))]
    [Export(typeof(IDockableScreen))]
    public class SceneGraphPanelViewModel : DockableScreen, IPlugin
    {
        private ObservableCollection<TreeNodeViewModel> _worldNodes;
        public ObservableCollection<TreeNodeViewModel> WorldNodes
        {
            get { return _worldNodes; }
        }

        public SceneGraphPanelViewModel()
        {
            _worldNodes = new ObservableCollection<TreeNodeViewModel>();
            this.DisplayName = "Scene Graph";
        }

        // IPlugin.Initialize() implementation
        public void Initialize()
        {
        }

        public void Exit()
        {
        }
    }
}
```

3. Add the following code to define methods for adding and removing objects from your tree node collections.

```
private WorldTreeNodeViewModel _3dWorldNode;
private WorldTreeNodeViewModel _drawingWorldNode;

// To keep the example simple this flag prevents the created component from being
// added to the tree, we will add it manually after the component has been built completely
private bool _isCreatingComponent;
...
private void ComponentAdded(object sender, ComponentAddedEventArgs e)
{
    if (!_isCreatingComponent)
    {
        _3dWorldNode.ComponentGroup.Children.Add(new SimTreeNodeViewModel(e.Component.RootNode));
    }
}

void ComponentRemoving(object sender, ComponentRemovingEventArgs e)
{
    RemoveComponentNode(_3dWorldNode, e.Component);
}

void LayoutItemAdded(object sender, LayoutItemAddedEventArgs e)
{
    _3dWorldNode.LayoutItemGroup.Children.Add(new LayoutItemTreeNodeViewModel(e.LayoutItem));
}

void LayoutItemRemoving(object sender, LayoutItemRemovingEventArgs e)
{
    RemoveLayoutItemNode(_3dWorldNode, e.LayoutItem);
}

void DrawingWorld_ComponentAdded(object sender, ComponentAddedEventArgs e)
{
    _drawingWorldNode.ComponentGroup.Children.Add(new SimTreeNodeViewModel(e.Component.RootNode));
}

void DrawingWorld_ComponentRemoving(object sender, ComponentRemovingEventArgs e)
{
    RemoveComponentNode(_drawingWorldNode, e.Component);
}

void DrawingWorld_LayoutItemAdded(object sender, LayoutItemAddedEventArgs e)
{
    _drawingWorldNode.LayoutItemGroup.Children.Add(new LayoutItemTreeNodeViewModel(e.LayoutItem));
}

void DrawingWorld_LayoutItemRemoving(object sender, LayoutItemRemovingEventArgs e)
{
    RemoveLayoutItemNode(_drawingWorldNode, e.LayoutItem);
}
```

```

private void RemoveComponentNode(WorldTreeNodeViewModel worldNode, ISimComponent component)
{
    TreeNodeViewModel treeNode = worldNode.ComponentGroup.
        Children.
        Cast<SimTreeNodeViewModel>().
        FirstOrDefault(n => n.Node == component.RootNode);

    if (treeNode != null)
    {
        worldNode.ComponentGroup.Children.Remove(treeNode);
    }
}

private void RemoveLayoutItemNode(WorldTreeNodeViewModel worldNode, ILayoutItem item)
{
    TreeNodeViewModel treeNode = worldNode.LayoutItemGroup.
        Children.
        Cast<LayoutItemTreeNodeViewModel>().
        FirstOrDefault(n => n.LayoutItem == item);

    if (treeNode != null)
    {
        worldNode.LayoutItemGroup.Children.Remove(treeNode);
    }
}

```

NOTE! A method signature must match the signature of an event handler in order to be added and executed when the event occurs during runtime.

4. Add the following code to add your methods to event handlers when the plugin is initialized during runtime.

```

...
[Import]
Lazy<IApplication> _application = null;
...

public void Initialize()
{
    IApplication app = _application.Value;

    app.World.ComponentAdded += ComponentAdded;
    app.World.ComponentRemoving += ComponentRemoving;
    app.World.LayoutItemAdded += LayoutItemAdded;
    app.World.LayoutItemRemoving += LayoutItemRemoving; ;

    app.DrawingWorld.ComponentAdded += DrawingWorld_ComponentAdded;
    app.DrawingWorld.ComponentRemoving += DrawingWorld_ComponentRemoving;
    app.DrawingWorld.LayoutItemAdded += DrawingWorld_LayoutItemAdded;
    app.DrawingWorld.LayoutItemRemoving += DrawingWorld_LayoutItemRemoving;

    _3dWorldNode = new WorldTreeNodeViewModel(app.World, "3D");
    _drawingWorldNode = new WorldTreeNodeViewModel(app.DrawingWorld, "Drawing");

    _worldNodes.Add(_3dWorldNode);
    _worldNodes.Add(_drawingWorldNode);
}

```

Creating View

The user interface of a plugin must use WPF and follow a naming convention to associate the logic of controls with a ViewModel.

1. In your project, add a **WPF User Control**, and name it **SceneGraphPanelView**.
2. Add a reference to **System.Xaml**.
3. In the XAML editor, add the following markup to define a TreeView control which is binded to your tree node collections.

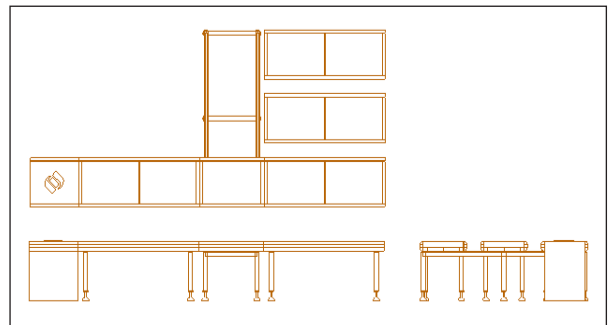
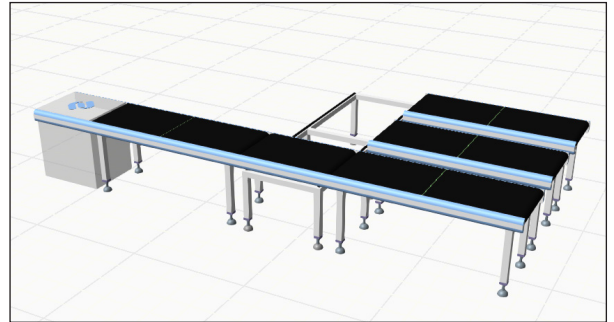
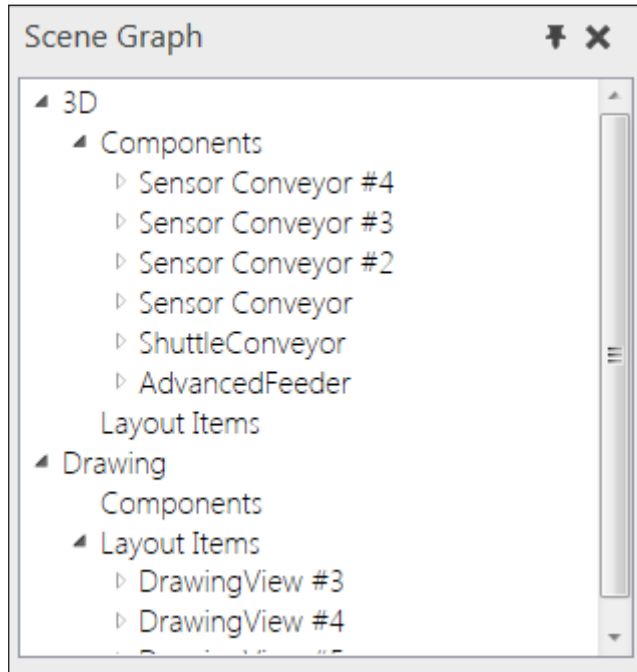
```
<UserControl x:Class="SceneGraphPanel.SceneGraphPanelView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="300"/>
            <RowDefinition Height="*/>
        </Grid.RowDefinitions>

        <TreeView Margin="5" Grid.Row="0" ItemsSource="{Binding WorldNodes}">
            <TreeView.ItemTemplate>
                <HierarchicalDataTemplate ItemsSource="{Binding Children}">
                    <TextBlock Text="{Binding Header}" />
                </HierarchicalDataTemplate>
            </TreeView.ItemTemplate>
        </TreeView>
    </Grid>
</UserControl>
```

Testing data collection and visualization

Your project is now in a state to be tested by adding and removing components from the 3D world and drawing of Essentials.

1. Start debugging your project. If you encounter any errors, handle those issues, and then resume debugging your project.
2. In Essentials, add a layout, create some drawings, delete one or more items in order to verify your scene graph panel plugin is working properly.



3. Stop debugging your project.

Creating components and other objects

In some cases, you may want to create components and other objects using API.

1. In the Code view for **SceneGraphPanelViewModel**, add the following code to define a method for creating a component that has a joint driven by a servo controller.

```
public void CreateComponent()
{
    _isCreatingComponent = true;

    ISimComponent component = _application.Value.World.CreateComponent("NewComponent");

    IProperty<int> length = (IProperty<int>)component.CreateProperty(typeof(int),
        PropertyConstraintType.AllValuesAllowed, "BlockLength");
    length.Value = 1000;

    IProperty<int> height = (IProperty<int>)component.CreateProperty(typeof(int),
        PropertyConstraintType.AllowedRangeOnly, "BlockHeight");
    height.AllowedMinimum = 100;
    height.AllowedMaximum = 1000;
    height.Value = 100;

    IProperty<int> width = (IProperty<int>)component.CreateProperty(typeof(int),
        PropertyConstraintType.AllowedValueSetOnly, "BlockWidth");
    width.AllowedValues.Add(new PropertyAllowedValue<int>() { Value = 100 });
    width.AllowedValues.Add(new PropertyAllowedValue<int>() { Value = 200 });
    width.AllowedValues.Add(new PropertyAllowedValue<int>() { Value = 300 });
    width.Value = 100;

    IBlockFeature blockFeature = component.RootNode.RootFeature.CreateFeature<IBlockFeature>();
    blockFeature.GetProperty("Length").Value = "BlockLength";
    blockFeature.GetProperty("Height").Value = "BlockHeight";
    blockFeature.GetProperty("Width").Value = "BlockWidth";

    ISimNode link1 = component.RootNode.AddChild("Link1");
    IDof dof = link1.Dof;
    dof.JointType = JointType.Linear;
    dof.JointAxis = JointAxis.PositiveX;
    dof.MinLimitExpression = "0";
    dof.MaxLimitExpression = "1000";

    IServoController servo = component.RootNode.CreateBehaviorByType(BehaviorType.ServoController) as IServoController;
    servo.Dofs.Add(dof);

    ITransformFeature transformFeature = link1.RootFeature.CreateFeature<ITransformFeature>();
    transformFeature.GetProperty("Expression").Value = "Tz(BlockHeight).Ty(BlockWidth/2)";

    ICylinderFeature cylinderFeature = transformFeature.CreateFeature<ICylinderFeature>();
    cylinderFeature.GetProperty("Height").Value = "100";
    cylinderFeature.GetProperty("Radius").Value = "50";
    link1.Material = _application.Value.Materials.FirstOrDefault(m => m.Name == "red");
    link1.MaterialInheritance = MaterialInheritance.ForceInheritNode;
```

```

INote note = component.RootNode.CreateBehaviorByType(BehaviorType.Note) as INote;
note.Note = "Press 'Interact' to reposition the cylinder";

component.Rebuild();

_isCreatingComponent = false;

_3dWorldNode.ComponentGroup.Children.Add(new SimTreeNodeViewModel(component.RootNode));
}

```

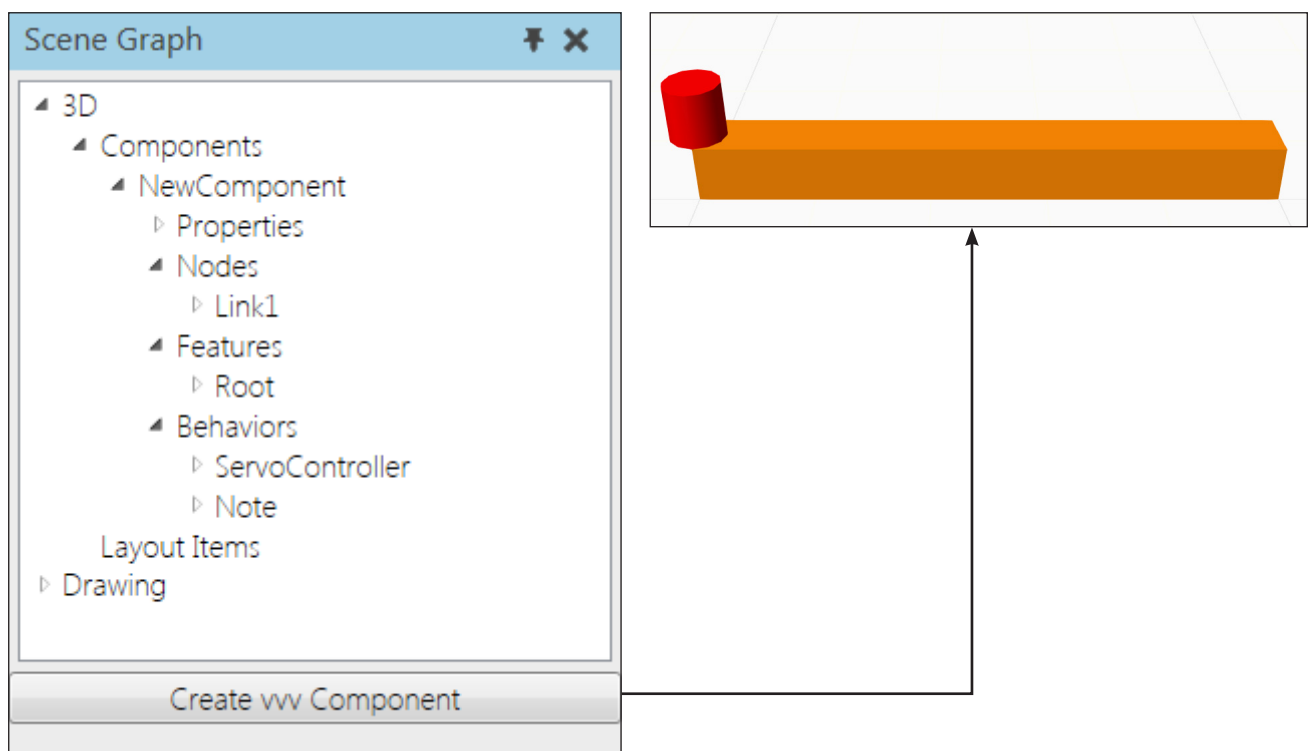
2. In the XAML editor for **SceneGraphPanelView**, add the following markup to create a button which is bound to the method used to create new components.

```

...
</TreeView>
<Button Grid.Row="1" VerticalAlignment="Top" x:Name="CreateComponent">Create vvv Component</Button>
</Grid>
</UserControl>

```

3. Start debugging your project.
4. In Essentials, verify the button in your plugin creates a component and its data structure is displayed in the TreeView control.



5. Stop debugging your project.

This concludes the tutorial.