ActiViz .NET User's Guide

Version 9.7

kitware

For More Information:

- Kitware provides training for VTK and ActiViz. Learn more at https://www.kitware.eu/training
- Kitware provides support and consulting services for ActiViz at https://www.kitware.eu/what-we-offer/#support
- Books explaining the theory and use of VTK are available from https://www.kitware.eu/what-we-offer/#books
- VTK examples and documentation

Contributors:

- Jeff Baumes — *Wikipedia Browser* example
- David Cole — Technical Lead
- Jon Crall — Technical Contributor, examples
- Bill Hoffman — Technical Contributor
- Niki Russell — Documentation, web support
- Will Schroeder — Documentation, examples
- Tristan Coulange — Technical Contributor
- Lucas Gandel — Technical Lead
- Alexy Pellegrini — Technical Contributor

and the world-wide VTK community at https://www.vtk.org.

Join the VTK Community at https://www.vtk.org

Table of Contents

1. Introduction

Welcome to the ActiViz .NET User's Guide.

This guide is organized into several parts: an introduction, an installation guide, core concepts and best practices, and an overview of the main environments in which ActiViz can be used.

The introduction provides a high-level overview of ActiViz .NET and its capabilities. The installation guide explains how to install and set up the software. Once ActiViz is installed, you can explore the examples distributed with the software and then continue with the Using ActiViz .NET section to familiarize yourself with its core concepts and prepare to write your own applications.

Finally, the last section provides links to additional resources and related information for readers who want to learn more.

1.1. What is ActiViz .NET?

ActiViz .NET provides an integration layer for The Visualization Toolkit (VTK), described in more detail in the following section, that enables VTK to be used within the Microsoft .NET ecosystem. This allows developers to leverage the capabilities of VTK using .NET programming languages such as C# and Visual Basic .NET.

ActiViz .NET is designed primarily for application developers building software in the Microsoft .NET environment. Although it includes advanced examples, realizing the full potential of ActiViz .NET requires developing software applications that make use of its VTK-based functionality.

1.2. What is VTK?

VTK, or the Visualization Toolkit, is an object-oriented software system for 3D graphics, data visualization, data processing, human-computer interaction, information visualization, volume rendering and much more. It has been under development for decades, and is used by researchers, developers and businesses from around the world. Tens of millions of dollars of labor have been invested in the system from commercial entities, government funding, and the open-source community. It is used for a diverse set of applications, including volume rendering, medical imaging (http://www.slicer.org), visualization (http://www.paraview.org), and many, many more.

1.3. How ActiViz .NET differs from VTK

VTK is an open-source system written in C++ that you can download, build, and use for free. However, using VTK requires significant C++ developer skills, and VTK does not easily

integrate into the Microsoft development environment. ActiViz .NET provides the appropriate integration layer so that VTK seamlessly fits into the .NET ecosystem. This means that you can use languages such as C# and Visual Basic to add powerful 3D visualization capabilities to your own applications. This integration layer provides the benefits of the .NET environment including on-line documentation and intelligent coding.

ActiViz interfaces with many existing applications and frameworks written in C#, including WindowsForm, Windows Presentation Foundation (WPF), WinUI 3, Avalonia, and the Unity software. This enables a seamless and fast integration of advanced algorithms and rendering techniques in various environments, Windows, Linux, MacOS and web browsers.

1.4. Licensing

While VTK is under the BSD 3-Clause License, ActiViz is proprietary and has its own license that allows you to redistribute ActiViz as part of your commercial application without having to purchase additional licenses for the end-user.

Read the ActiViz license agreement or contact us for more information.

2. Getting ActiViz .NET

Request a trial version of ActiViz .NET here: https://www.kitware.eu/product/activiz
You can find details about the latest version of ActiViz and its pricing at this link : https://www.kitware.eu/get-activiz.

Notes:
The latest “Open-Source” version available is ActiViz .NET 5.8.0.
The latest “Supported” version available is ActiViz .NET 9.7.

3. Installing ActiViz .NET

This section describes how to download and install ActiViz .NET and configure Microsoft Visual Studio to use ActiViz .NET on Windows.

Packages for Linux and macOS are provided as archives. After installing the .NET SDK, these packages can be made available locally to the NuGet command-line tools. For more information, see the Command Line section.

For instructions on installing the .NET SDK for your operating system, see the Microsoft .NET download page.

3.1. System Requirements

ActiViz .NET runs on Windows 10 to Windows 11, GNU Linux, OSX 13+ and all modern web browsers.
While VTK itself can run on small to large computers, it is a sophisticated, powerful system that requires adequate computing resources. The following hardware configuration is recommended for optimal performance when using ActiViz .NET:

- **Memory (RAM)**: **8 GB** or more is recommended. As a general guideline, system memory should be approximately 10 times the amount of data being loaded into memory.
- **Graphics (GPU)**: A dedicated graphics processing unit (discrete GPU) is recommended for fast rendering. Integrated graphics are sufficient for basic visualization but are not recommended for demanding workloads.
    - Discrete GPUs, such as NVIDIA GPUs, are recommended.
    - The GPU must support **OpenGL 3.2** or later.
    - **4 GB or more of VRAM** is recommended. For large datasets, the available GPU memory shall be sufficient to accommodate the largest dataset or the working set required for rendering.
- **Processor (CPU)**: Most calculations are performed on the CPU and can benefit from multi-core and multi-CPU configurations. A modern **Intel Core i7 or i9** processor, or equivalent, is recommended.
- **Display**: A resolution of **1280 × 1024** or higher is recommended.

3.2. Installation With Installer (Windows)

Once you have downloaded the appropriate installer, open it and follow the instructions. The installation wizard guides you through a series of steps to configure and install ActiViz .NET on your Windows system.
The installation process proceeds as follows:

- **Welcome**: The installer displays a welcome screen. Select “Next >” to continue.
- **License Agreement**: Review and accept the license agreement to proceed.
- **Installation Location**: Select the destination folder where ActiViz .NET will be installed, or use the default location.

⚠️**WARNING:** Installing ActiViz .NET in a directory that requires administrator privileges, such as **Program Files**, may cause runtime issues that require your

development environment to run with elevated (administrator) privileges.
To avoid these issues, we recommend installing ActiViz .NET in a directory where the user has full read and write access and does not require administrator privileges.

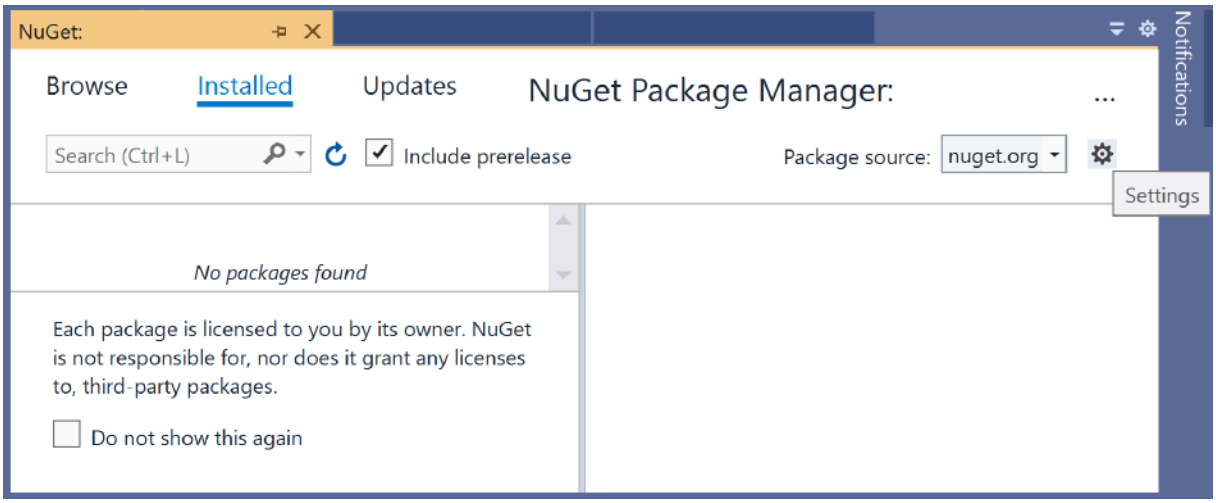
- **Desktop Shortcut**: Choose whether to create a desktop shortcut for ActiViz .NET.
- **Install**: Review the selected settings, then select **Install** to begin the installation. The installer copies the required files and configures ActiViz .NET on your system.
- **Completion**: Once the installation is complete, select **Finish** to exit the installer

3.3. Installation With NuGet

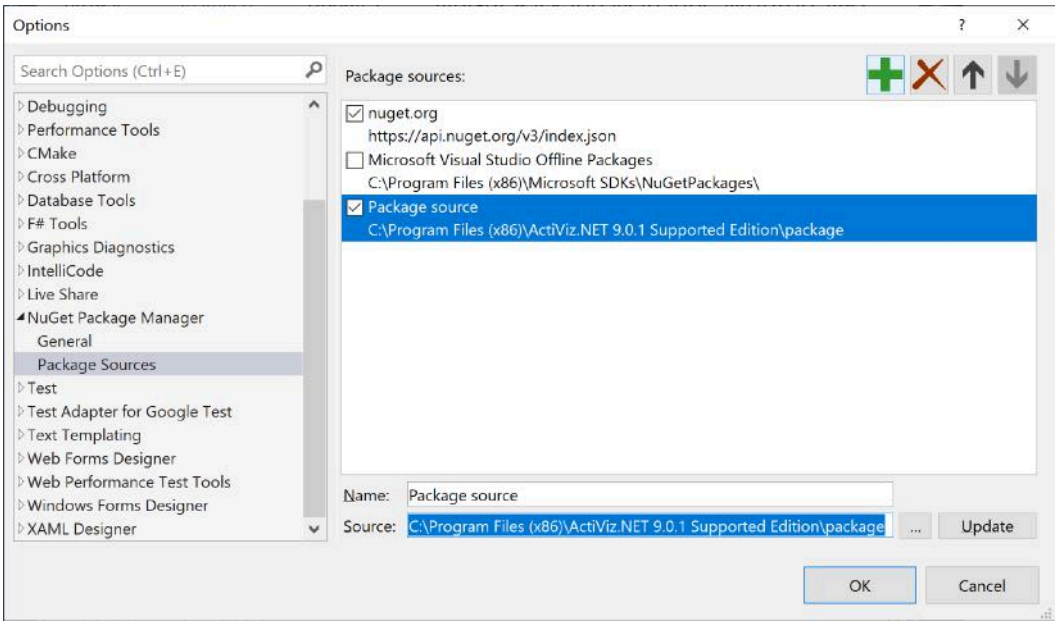
ActiViz Nuget packages of the latest Supported version targeting .NET Core are provided with the installer (Windows) and archives (Linux, Mac, web browsers).
To install and use ActiViz in .NET Core projects, you must provide NuGet with the location of the packages that come with the installer. This can be achieved by adding the **package** directory, located in the ActiViz installation folder, as a local NuGet package source, either through Visual Studio or directly from the command line as described below.

❖ Visual Studio

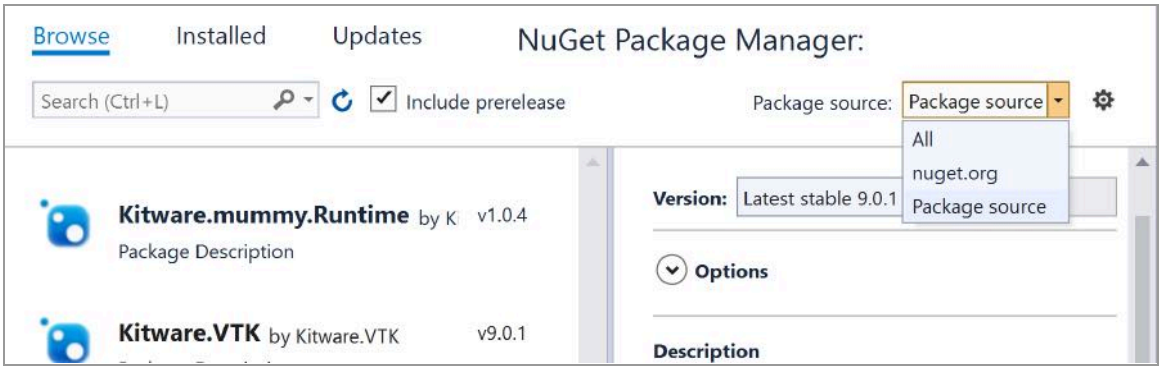
- Right-click on your project
- Select “Manage NuGet Packages”
- Click the Package Source Settings wheel in the upper right corner of the Nuget Package Manager



- Add a new package source using the “+” icon, and update the Source path to include the location of your ActiViz nuget packages



- The new package source appears in the list and allows for browsing local packages. You can now install Kitware.VTK and start using ActiViz.



❖ Command line

On any system, one can add a package source using the following command:

```
dotnet nuget add source “path/to/ActiViz/package”
```

After this has been called, dotnet will be able to automatically find the ActiViz NuGet packages when building a project that depends on Kitware.VTK or other packages.

To add an ActiViz .NET package to your project, use the `dotnet add package` command and specify the package name. You can also specify the local NuGet source in which the package is available using the `--source` option. For example:

```
dotnet add package <Name> --source "<path-to-ActiViz>\package"
```

This adds the package as a dependency in your project and updates the project file accordingly. Once the package reference has been added, the ActiViz .NET assemblies and associated dependencies are available to the application.

To build and run a project, use the following command from the directory containing the `.csproj` project file:

```
dotnet run
```

4. Using ActiViz .NET

4.1. ActiViz .NET Core Nuget Package (Recommended)

Starting at version 9.1, NuGet packages of ActiViz are provided for .NET Core applications.
The following target frameworks are available within the package:

- net10
- net10-windows (For WindowsForm and WPF support)

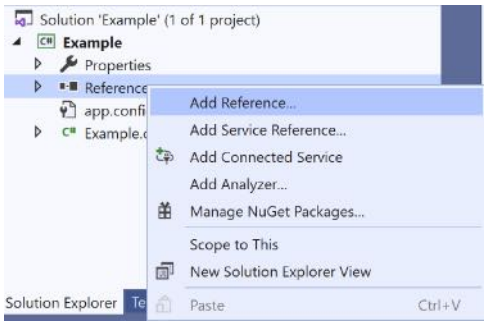
To start using ActiViz in .NET Core applications, follow both the installation instructions of sections Installation With Installer and Installation With Nuget.

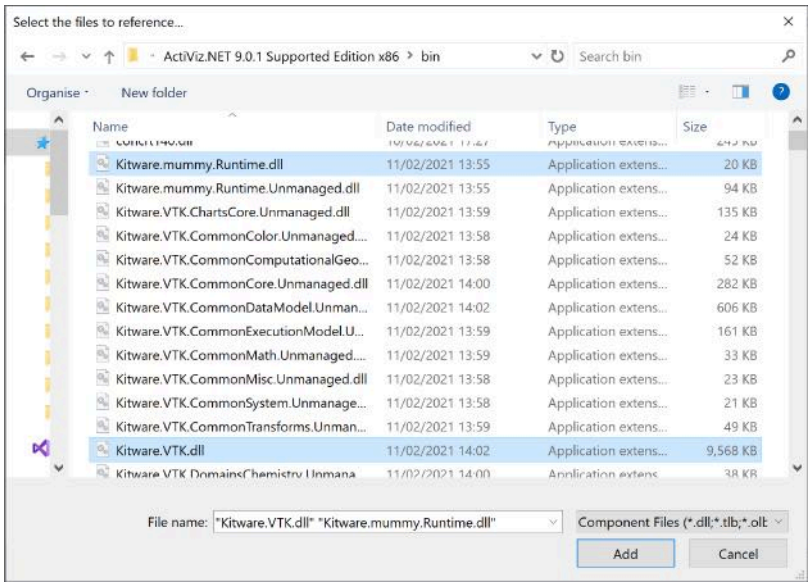
4.2. ActiViz .NET Framework Reference

Assembly references can be used on Windows for projects targeting the .NET Framework. To use ActiViz in your project, the first step is to add references to the ActiViz libraries.

In Solution Explorer, right-click the project and select Add Reference. In the dialog that opens, select Browse and navigate to the ActiViz .NET installation directory. Then select the two assemblies shown in the figure below.

- “Kitware.mummy.Runtime.dll”
- “Kitware.VTK.dll”.





4.3. A brief overview of VTK

Though VTK is a large and complex software system, knowledge of its conceptual framework will greatly assist you in understanding the basics of ActiViz.

VTK is a data-centric visualization toolkit built around a data-flow pipeline. Data moves through a sequence of processing stages, where each stage transforms the data before passing it to the next.

- ❖ **Processing Pipeline**

The processing pipeline is constructed by connecting algorithms, also known as process objects. Each algorithm performs a specific operation on the input data and produces output that can be consumed by another algorithm.

For example, a typical pipeline might:

- Read polygonal data from a file.
- Decimate the geometry to reduce its complexity.
- Smooth the resulting geometry.
- Pass the processed data to the rendering subsystem.

Algorithms are connected using the `SetInputConnection()` and `GetOutputPort()` methods. The VTK pipeline is automatically updated when the last algorithm in the pipeline requests an update. Any algorithm that is not up to date is executed to produce its output,

which is then passed to the next filter. Update requests therefore cascade from the end of the pipeline toward the beginning, while data processing proceeds in the opposite direction, from the beginning toward the end.
Alternatively, algorithms can be connected using `SetInputData()` and `GetOutput()`. This passes data directly between algorithms rather than creating a pipeline connection. In this case, each algorithm must be updated explicitly by calling its `Update()` method.

This pipeline-based architecture makes it possible to combine and reuse individual processing algorithms to build more complex visualization workflows. VTK can also be used purely as a data-processing engine, without rendering anything, for example to load data, transform or analyze it, and write the processed result back to disk.

- ❖ **Visualization and Rendering**

The visualization pipeline takes the output of the processing pipeline and prepares it for display. Its primary role is to transform processed data into rendering primitives that can be consumed by VTK's graphics subsystem.

Conceptually, the flow is:

Input data → Processing pipeline → Visualization pipeline → Display

The rendering stack then combines these primitives with scene information such as cameras, lights, and material properties to produce the final 2D image displayed on the screen.

When rendering the results of a pipeline, the last algorithm is typically a VTK mapper, which converts the processed data into graphics primitives suitable for rendering. The mapper is automatically updated when its associated actor is requested to render.
Unlike many UI frameworks, VTK does not rely on a continuous render loop with a fixed tick interval. Rendering occurs when the pipeline has been modified, or when an explicit render request is made. Render requests can be issued at any time by calling the `Render()` method on the render window interactor.

VTK's rendering system represents a 3D scene using several key objects:

- **`vtkActor / vtkProp`** - Represent objects that appear in the scene and are rendered. vtkActor is a subclass of vtkProp; the term prop follows the theatrical notion of objects placed on a stage. In general, rendered scene objects are often referred to as actors.
- **`vtkCamera`** - Defines the viewpoint and projection used to transform the 3D scene into a 2D image.
- **`vtkLight`** - Provides illumination for the scene.
- **`vtkProperty`** - Defines the visual and material characteristics of rendered objects, including properties that affect lighting and appearance.

- **`vtkRenderer`** - Defines the rendered area and manages the rendering of a scene from a particular viewpoint to produce the rendered image.
- **`vtkRenderWindow`** - Provides the final rendering surface and can contain one or more renderers, allowing multiple views or rendering regions to be combined in a single window.

These objects follow the familiar lights–camera–actors model used in filmmaking: actors make up the scene, lights illuminate them, and a camera determines how the scene is viewed.
Behind this high-level model, VTK contains many additional components that support the rendering process. These include `vtkTransform` objects for geometric transformations, interactors for handling mouse and keyboard input, and `vtkTexture` objects for applying texture maps to rendered geometry.

Overall, VTK can therefore be viewed as two closely connected layers: a data-processing pipeline, which transforms and prepares data, and a rendering stack which converts that data into a visual scene and ultimately produces the image displayed to the user.
For a complete example showing how to produce and render data with ActiViz, check the Console Application section.

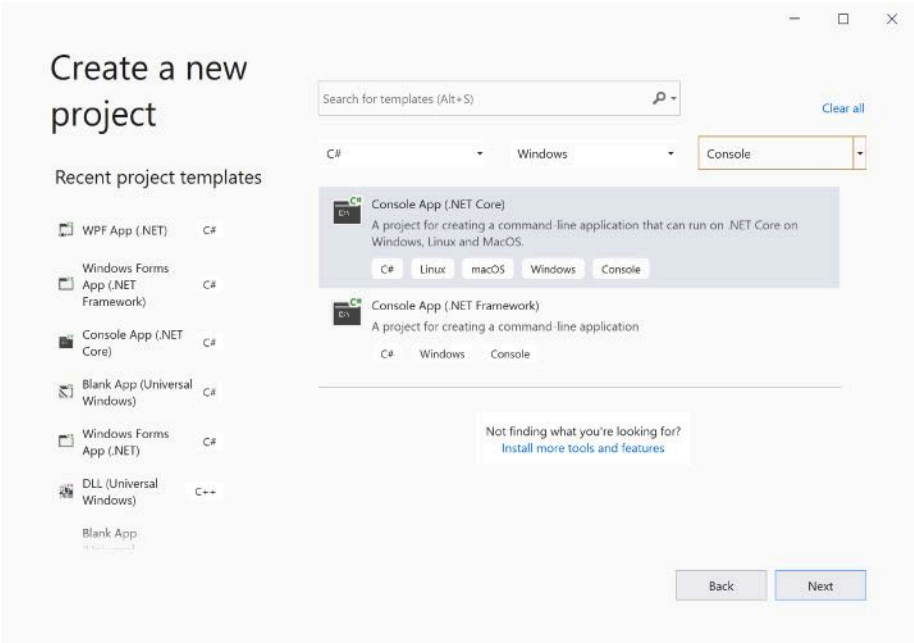
❖ User interactions

The key classes for handling mouse and keyboard events in a render window are **`vtkRenderWindowInteractor`** and **`vtkInteractorStyle`**. VTK internally uses an event/observer pattern, which allows specific behaviors to be associated with user interactions. These behaviors can be used to control the scene, for example to rotate or translate the camera, zoom in or out, or to trigger custom actions in response to user input.
By default, VTK provides dozens of `vtkInteractorStyle` subclasses, each implementing a specific set of interaction behaviors suited to different use cases. This allows applications to select the interaction style that best matches the desired user experience or to extend an existing style with custom behavior.
To define custom behaviors in response to events, read the VTK events and observers section.

Besides the core classes described here, VTK provides hundreds of additional classes that implement key functionality across the toolkit. These include filters for data processing, interaction widgets for direct manipulation of data and graphical elements, image-processing algorithms, volume-rendering techniques, information visualization, mathematical operations, computational geometry, and much more.

4.4. Console application

Console applications provide the quickest and simplest way to build and experiment with VTK pipelines in ActiViz. They have no user interface overhead, allowing developers and researchers to focus entirely on processing and rendering logic. This makes console applications ideal for rapidly prototyping, testing new processing or rendering pipelines, and conducting research where fast iteration and direct interaction with VTK are more important than building a complete graphical application.
For more advanced use cases and production-ready applications, see the next section on using UI frameworks.
To set up a console application, create a new Visual Studio project and choose the Console App template.



Alternatively, you can adapt the `HelloVTKConsole` example provided in the Examples folder of your ActiViz installation directory, or use the .NET CLI to create a project directly from the command line:

```
dotnet new console
```

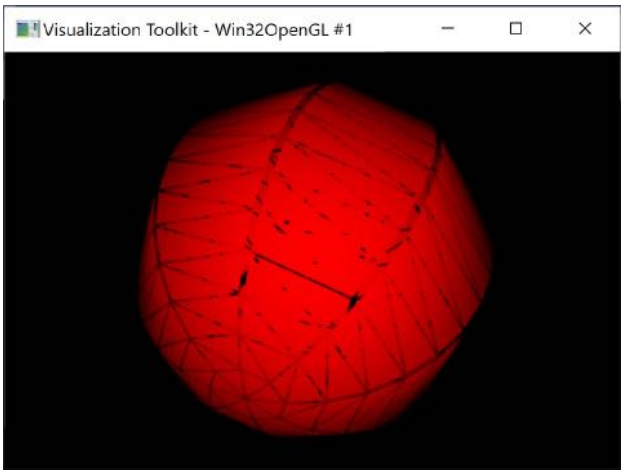
Open the created project and add the required references to the ActiViz **Kitware.VTK** package by following the instruction in Using ActiViz .NET.

You can now add the appropriate VTK code to the `Main()` function. Note the `using Kitware.VTK` namespace directive at the top of the file, which provides access to the VTK classes available in ActiViz.

CSharp

```
using System;
using System.Collections.Generic;
using System.Text;
using Kitware.VTK;

namespace HelloVTK
{
  class Program
  {
    static void Main(string[] args)
    {
      // Create a simple sphere and plug it to the shrink polyData filter.
      // A pipeline is created.
      var sphere = vtkSphereSource.New();
      sphere.SetThetaResolution(8);
      sphere.SetPhiResolution(16);

      var shrink = vtkShrinkPolyData.New();
      shrink.SetInputConnection(sphere.GetOutputPort());
      shrink.SetShrinkFactor(0.9);

      // The mapper links the data pipeline to the rendering pipeline.
      // The actor represents the object in the scene
      var mapper = vtkPolyDataMapper.New();
      mapper.SetInputConnection(shrink.GetOutputPort());

      var actor = vtkActor.New();
      actor.SetMapper(mapper);
      actor.GetProperty().SetColor(1,0,0);

      // Create components of the rendering subsystem
      var renderer = vtkRenderer.New();
      var renderWindow = vtkRenderWindow.New();
      renderWindow.AddRenderer(renderer);
      var interactor = vtkRenderWindowInteractor.New();
      interactor.SetRenderWindow(renderWindow);

      // Add the actor to the renderer and start the event loop
      renderer.AddViewProp(actor);

      interactor.Start();
    }
  }
}
```

Compiling and running the above C# program produces the red sphere shown below. The example creates a simple VTK pipeline that generates polygonal data using a sphere source, shrinks the polygons toward their centers using a shrink filter, and then maps the resulting data to graphics resources for rendering.



In console applications, the program event loop is managed by the native VTK render window interactor. The call to `interactor.Start();` at the end of the example is a blocking call that starts the VTK event loop and handles user input events. When the event loop runs for the first time, the render window is initialized and a render request is triggered automatically, causing the pipeline to be updated and rendered.

By default, VTK also creates a camera and lights automatically in the vtkRenderer when none are explicitly provided by the application.

4.5. UI Frameworks

ActiViz supports a variety of UI frameworks for developing production-ready applications across different platforms. In addition to the `Kitware.VTK` NuGet package, which provides access to the full VTK API from C#, ActiViz provides dedicated NuGet packages for seamlessly integrating VTK into supported UI frameworks through custom controls called `RenderWindowControl`.

`RenderWindowControl` classes derive from the framework's `Control` class and expose a `vtkRenderWindow` and a `vtkRenderWindowInteractor` instance. They bridge VTK and the UI framework by forwarding user input events from the framework to the render window interactor and integrating the rendered output of the VTK render window into the application's UI.

For a complete list of supported UI frameworks and platforms, check the Supported platforms

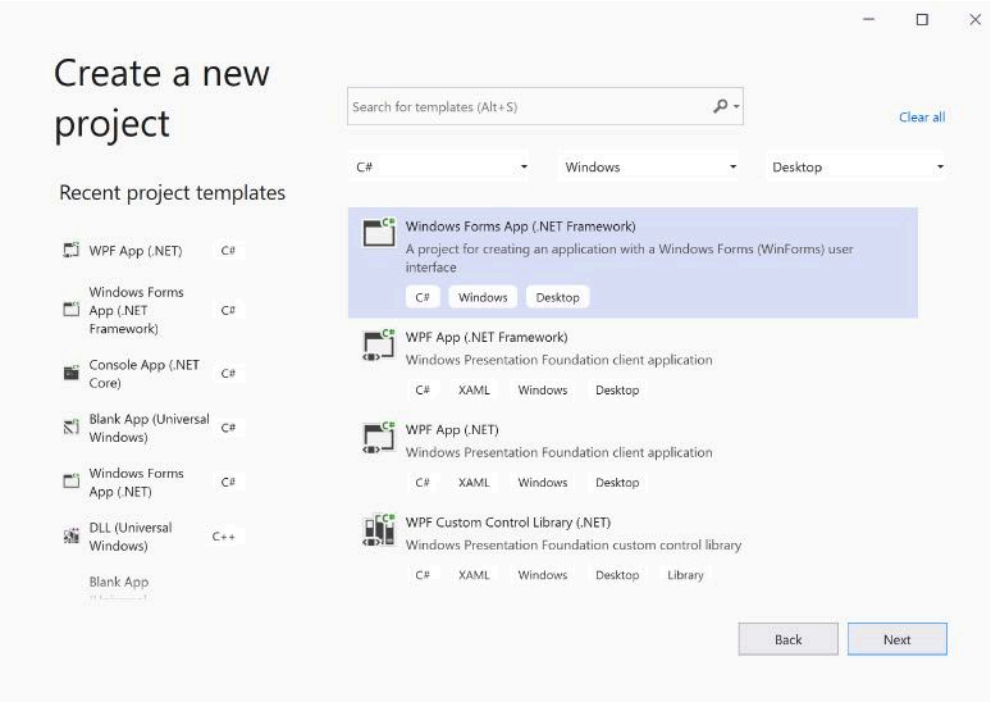
section. The UI frameworks most commonly used with ActiViz are described in the subsections below.

4.5.1. Windows Form

Windows Forms (WinForms) is the first and historically most widely used UI framework with ActiViz. It is a mature, Windows-specific framework that remains well suited to traditional desktop applications and existing WinForms codebases. ActiViz provides a dedicated `RenderWindowControl` for seamlessly integrating VTK rendering and interaction into WinForms applications. While newer cross-platform UI frameworks may be preferable for new applications, WinForms remains a simple and well-established choice for Windows desktop applications.

- **Create a new project**

To set up a Windows Form application, create a new Visual Studio project and choose the Windows Form App template.



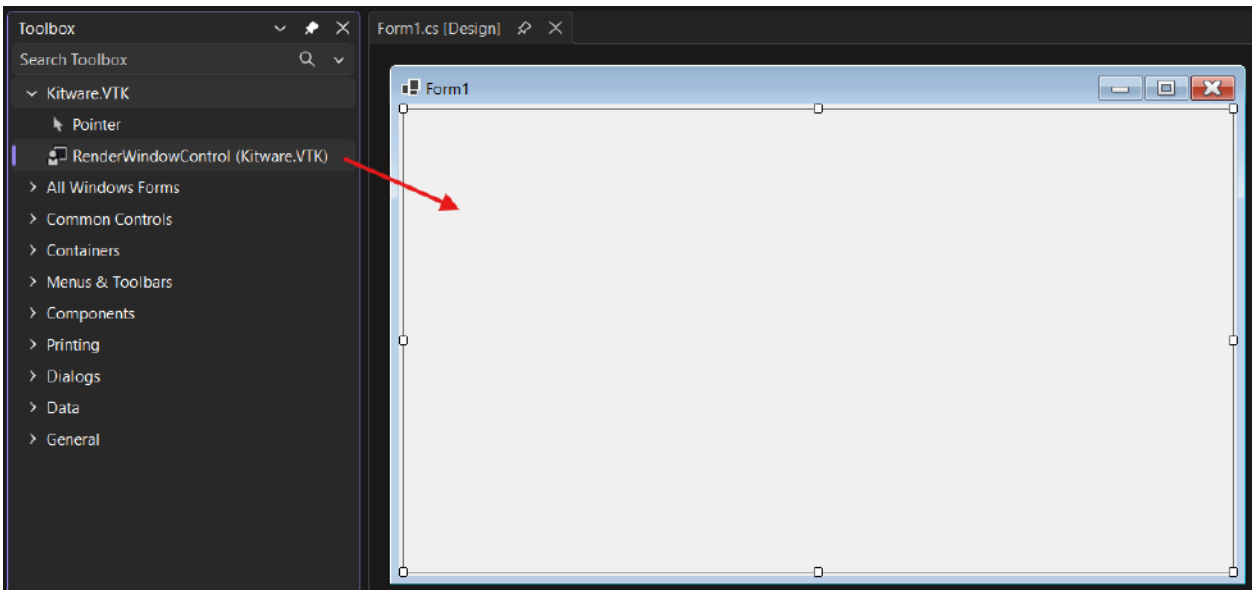
Alternatively, you can adapt the `HelloVTKForm` example provided in the Examples folder of your ActiViz installation directory, or use the .NET CLI to create a project directly from the command line:

```
dotnet new winforms
```

Open the created project and add the required references to the ActiViz package by following the instruction in Using ActiViz .NET. Up to ActiViz 9.7, the `RenderWindowControl` class for Windows Form is shipped as part of the `Kitware.VTK` assembly. More recent versions ship it in the `Kitware.WinFormsControls` NuGet package.

- **Initialize the VTK scene**

After adding a reference to the appropriate ActiViz NuGet package, the `RenderWindowControl` class becomes available in the Visual Studio Designer. From the **View** menu, make sure the **Toolbox** is visible. Locate `RenderWindowControl` in the Toolbox and drag it onto the form, as shown in the figure below.



Double-click the newly added control to automatically create a handler for its `Load` event and open the code editor.
The generated `renderWindowControl1_Load` event handler provides a convenient entry point for initializing your VTK resources. At this point, the control has already initialized the underlying render window and interactor, which are available for use by your application.

You can then populate the VTK render window in the same way as in the console application example, with one important difference: the render window and interactor are already provided by the `RenderWindowControl`.

⚠️**WARNING:** Do not start the VTK interactor event loop by calling `Start()`. The event loop is already managed by the Windows Forms application, and the `RenderWindowControl` automatically forwards the relevant UI events to the VTK interactor.

The resulting code is shown below:

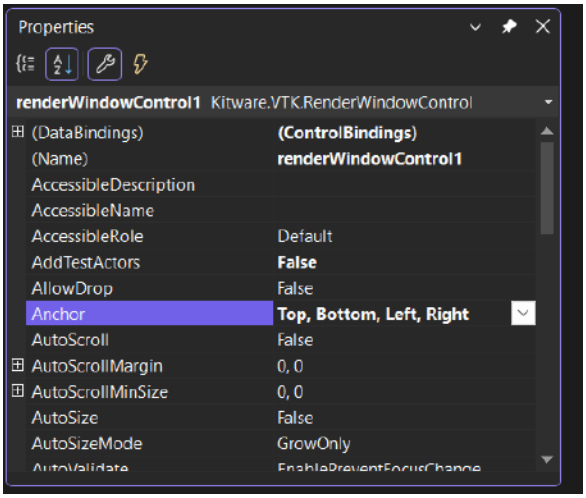
CSharp

```
using Kitware.VTK;

namespace HelloVTK;

public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }

  private void renderWindowControl1_Load(object sender, EventArgs e)
  {
    // Create a simple sphere and plug it to the shrink polyData filter.
    // A pipeline is created.
    var sphere = vtkSphereSource.New();
    sphere.SetThetaResolution(8);
    sphere.SetPhiResolution(16);

    var shrink = vtkShrinkPolyData.New();
    shrink.SetInputConnection(sphere.GetOutputPort());
    shrink.SetShrinkFactor(0.9);

    // The mapper links the data pipeline to the rendering pipeline.
    // The actor represents the object in the scene
    var mapper = vtkPolyDataMapper.New();
    mapper.SetInputConnection(shrink.GetOutputPort());
```

```
    var actor = vtkActor.New();
    actor.SetMapper(mapper);
    actor.GetProperty().SetColor(1, 0, 0);

    // Add a renderer to the control's render window
    var renderer = vtkRenderer.New();
    renderWindowControl1.RenderWindow.AddRenderer(renderer);

    // Add the actor to the renderer
    renderer.AddViewProp(actor);
  }
}
```

💡 To ensure that the `RenderWindowControl` is automatically resized when the form is resized, modify its `Anchor` property to `Top, Bottom, Left, Right`. This will automatically add the following code in the Form1.Designer.cs file:

```
renderWindowControl1.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
```



4.5.2. Windows Presentation Foundation - WPF

WPF (Windows Presentation Foundation) is a modern alternative to WinForms and has been the recommended UI framework for ActiViz applications targeting Windows for many years. ActiViz provides two approaches for integrating VTK into WPF applications.

- **WPF-WindowsForm Interoperability**

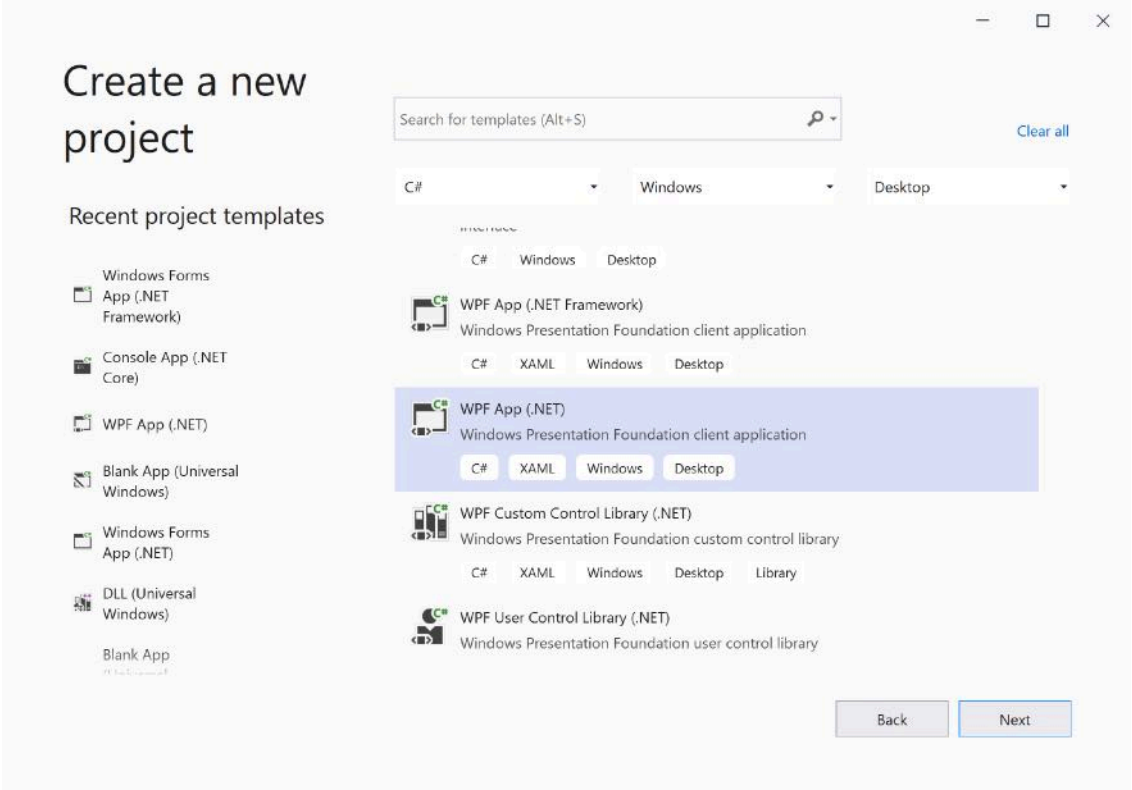
The first approach leverages WPF–WinForms interoperability through the `WindowsFormsHost` control. This allows the WinForms `RenderWindowControl` provided by ActiViz to be embedded directly into a WPF application. This approach provides the simplest way to integrate VTK into WPF, but it inherits some limitations of WPF–WinForms interoperability. In particular, certain layout limitations can affect the application design, such as preventing the control from being freely rotated and causing *airspace* issues when WPF elements overlap the hosted WinForms control.

- **WPF native control**

The second approach uses a native WPF `Image` control together with Microsoft's `WPFDXInterop` package to present the VTK rendering output directly within the WPF visual tree. Because this approach uses a native WPF control, it avoids the layout limitations associated with `WindowsFormsHost`. However, integrating VTK using this approach requires more advanced rendering considerations, particularly around OpenGL–DirectX interoperability. Although OpenGL–DirectX interoperability should be supported by all recent graphics drivers, this approach relies on a more advanced graphics technology stack and may therefore have compatibility limitations on older hardware or graphics drivers.

- **Create a new project**

To set up a WPF application, create a new Visual Studio project and choose the WPF App template.



Alternatively, you can adapt the `HelloVTKWPF` and `HelloVTKWPF_Native` examples provided in the Examples folder of your ActiViz installation directory, or use the .NET CLI to create a project directly from the command line:

```
dotnet new wpf
```

Open the created project and add the required references to the ActiViz package of your choice by following the instruction in Using ActiViz .NET.
To use the WindowsFormHost approach, add a reference to the ActiViz NuGet package presented in the Windows Form section.
To use the WPF native approach, add a reference to the `Kitware.WPFControls` NuGet package.

- **Initialize the VTK scene**

Edit the `MainWindow.xaml` description to add the ActiViz .NET `RenderWindowControl`. The assembly and namespace defined in the xmlns definitions determines which version of the `RenderWindowControl` is used.

❖ **WPF-WindowsForm Interoperability**

```
<Window x:Class="HelloVTKWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:vtk="clr-namespace:Kitware.VTK;assembly=Kitware.VTK"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <WindowsFormsHost Grid.Column="0" Grid.RowSpan="1" Name="wfh">
            <vtk:RenderWindowControl x:Name="VTKControl" Load="Form_Loaded" />
        </WindowsFormsHost>
    </Grid>
</Window>
```

Similar to the Windows Form example, add an event handler to the WinForm control `Load` event to populate the VTK scene in the `MainWindow.xaml.cs` file.

CSharp

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  private void Form_Loaded(object sender, EventArgs e)
  {
    InitVTKScene(VTKControl.RenderWindow);
  }

  private void InitVTKScene(vtkRenderWindow renderWindow)
  {
    ...
  }
}
```

❖ **WPF native control**

MainWindow.xaml

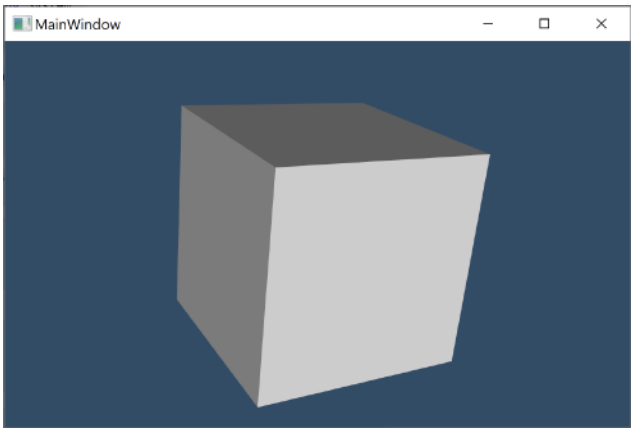
```xaml
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:vtk="clr-namespace:Kitware.WPFControls;assembly=Kitware.WPFControls"
        mc:Ignorable="d"
        Title="HelloVTKWPF-Native" Height="450" Width="800">

    <Grid>
      <vtk:RenderWindowControl Grid.Column="0" x:Name="VTKControl" Stretch="Fill" MinWidth="1" MinHeight="1"/>
    </Grid>
</Window>

```

When using the native approach, the scene can be initialized directly in the constructor of the main window.

CSharp

```csharp
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    InitVTKScene(VTKControl.RenderWindow);
  }

  private void InitVTKScene(vtkRenderWindow renderWindow)
  {
    ...
  }
}
```

⚠️**WARNING:** When using the native `RenderWindowControl` approach, explicitly request rendering by calling `Render()` on the render window interactor. This ensures that the rendering resources are properly synchronized. Calling `Render()` directly on the render window has no effect.



The initialization code common to both approaches is given below:

```CSharp
private void InitVTKScene(vtkRenderWindow renderWindow)
{
  // Create a simple cube.
  vtkCubeSource cube = vtkCubeSource.New();

  vtkPolyDataMapper mapper = vtkPolyDataMapper.New();
  mapper.SetInputConnection(cube.GetOutputPort());

  vtkActor actor = vtkActor.New();
  actor.SetMapper(mapper);

  // Add renderer to the render window
  vtkRenderer renderer = vtkRenderer.New();
  renderer.SetBackground(.2, .3, .4);
  renderWindow.AddRenderer(renderer);

  // Add the actor to the renderer and make it fit the viewport
  renderer.AddActor(actor);
  renderer.ResetCamera();
}
```

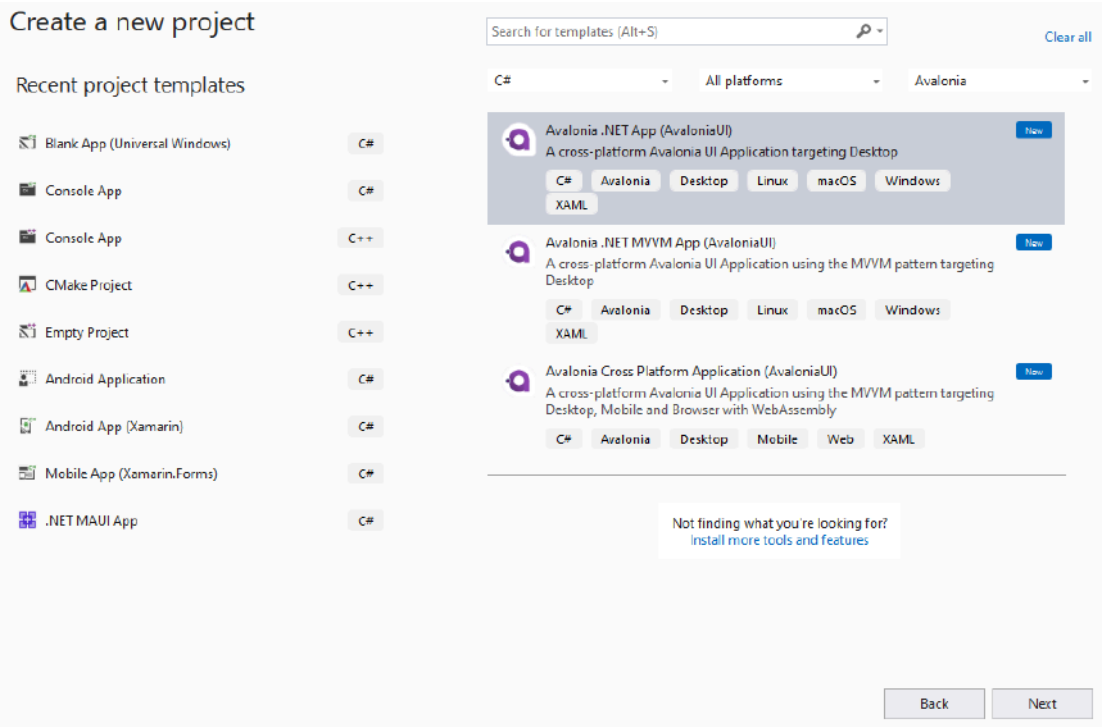
4.5.3. Avalonia

Avalonia is the recommended UI framework for building cross-platform applications with ActiViz. It is a modern, actively maintained framework based on XML UI description, making it easier to port existing Windows-only WPF applications to a cross-platform environment.

❖ **Create a new project**

To create an Avalonia application, install the Avalonia templates using the following command: `dotnet new install Avalonia.Templates`.

Then, select "Create a new project" from the Visual Studio start menu. Select an Avalonia App template and choose the project name and location.



Alternatively, you can adapt the `HelloVTKAvalonia` example provided in the Examples folder of your ActiViz installation directory

Once the solution is loaded, add the required `Kitware.AvaloniaControls` NuGet package to the list of dependencies by following the instruction in Using ActiViz .NET Core Nuget Package. The `Kitware.VTK` and `Kitware.mummy.Runtime` packages are automatically added to the project as a dependency of `Kitware.AvaloniaControls`.

❖ Rendering backend properties

As VTK's rendering is based on OpenGL, the Avalonia rendering backend must be explicitly configured to use OpenGL, as it is not the default backend. To do so, add the following options to the AppBuilder configuration in the `Program.cs` file.

```CSharp
public static AppBuilder BuildAvaloniaApp()
  => AppBuilder.Configure<App>()
     .UsePlatformDetect()
     .With(new Win32PlatformOptions { RenderingMode = new[] { Win32RenderingMode.Wgl } })
     .With(new X11PlatformOptions { RenderingMode = new[] { X11RenderingMode.Glx } })
     .With(new AvaloniaNativePlatformOptions { RenderingMode = new[] {
AvaloniaNativeRenderingMode.OpenGl } });
```

❖ Initialize the VTK scene

To add the ActiViz `RenderWindowControl` to the application window, open the `MainWindow.axaml` file, and edit the xml to define the namespace and add the control to the window as highlighted below:

MainWindow.axaml

```
1   <Window xmlns="https://github.com/avaloniaui"
2           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
3           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
4           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
5           xmlns:vtk="clr-namespace:Kitware.AvaloniaControls;assembly=Kitware.AvaloniaControls"
6           xmlns:ctrl-pages="clr-namespace:ControlCatalog.Pages"
7           mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
8           x:Class="HelloVTKAvalonia.MainWindow"
9           Title="HelloVTKAvalonia">
10
11    <vtk:RenderWindowControl Name="VTKControl"/>
12
13  </Window>
14
```

Next, add an event handler for the `RenderWindowControl`'s `AttachedToVisualTree` event. This handler will be used to initialize the VTK scene. Update the `MainWindow.xaml.cs` code as follows to register the `AttachedToVisualTree` event handler and initialize the scene in the corresponding callback:

CSharp

```
using Avalonia.Controls;
using Avalonia.Markup.Xaml;
using Kitware.AvaloniaControls;
using Kitware.VTK;
using System;

namespace HelloVTKAvalonia
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }

    private void InitializeComponent()
    {
      AvaloniaXamlLoader.Load(this);
      RenderWindowControl ctrl =
        this.FindControl<RenderWindowControl>("VTKControl");
      if (ctrl != null)
      {
        ctrl.AttachedToVisualTree += InitVTKScene;
      }
    }

    public void InitVTKScene(object? sender, EventArgs args)
    {
      RenderWindowControl? mainView = sender as RenderWindowControl;
      vtkRenderer renderer = vtkRenderer.New();
      renderer.SetBackgroundAlpha(1.0);
      mainView.RenderWindow.AddRenderer(renderer);

      vtkInteractorStyleTrackballCamera interactorStyle =
        vtkInteractorStyleTrackballCamera.New();
      renderWindow.GetInteractor().SetInteractorStyle(interactorStyle);
```

```
      vtkSphereSource src = vtkSphereSource.New();
      vtkPolyDataMapper mapper = vtkPolyDataMapper.New();
      mapper.SetInputConnection(src.GetOutputPort());
      vtkActor actor = vtkActor.New();
      actor.SetMapper(mapper);
      renderer.AddActor(actor);
    }
  }
}
```

⚠️**WARNING:** Explicitly request rendering by calling `Render()` on the render window interactor. This ensures that the rendering resources are properly synchronized. Calling `Render()` directly on the render window is unsupported.

❖ Threading model

Avalonia uses a threading model where UI-related work happens on the UI thread, while rendering is handled separately by the compositor.
Although VTK objects can be accessed and manipulated from both threads, the VTK rendering context is active only on the compositor thread. Consequently, any VTK operation that requires the OpenGL context to be current must be executed on the compositor thread to ensure stable behavior. This includes any operation that ultimately invokes OpenGL functions in the underlying VTK C++ code, such as rendering, render-based (hardware) picking, or removing actors from a renderer.

The `RenderWindowControl` implementation already provides mechanisms to ensure that several common operations are executed on the compositor thread. These include render requests as well as mouse and keyboard event handlers.

Running user input handlers on the compositor thread is particularly useful for interactions that require access to the rendering context, such as VTK hardware picking. Thanks to this approach, for example, users can perform hardware picking directly from mouse event handlers without having to explicitly dispatch their code to the compositor thread:

```CSharp
renderWindow.GetInteractor().RightButtonPressEvt += OnRightButtonPress;
...
private void OnRightButtonPress(vtkObject sender, vtkObjectEventArgs e)
{
  vtkRenderWindowInteractor iren = vtkRenderWindowInteractor.SafeDownCast(sender);
  vtkRenderer ren = iren.GetRenderWindow().GetRenderers().GetFirstRenderer();
  vtkPropPicker picker = vtkPropPicker.New();
  int[] mousePos = iren.GetEventPosition();
  int p = picker.Pick(mousePos[0], mousePos[1], 0, ren);
  Console.WriteLine("Pick successful: " + p);
}
```

In addition to these built-in mechanisms, `RenderWindowControl` provides the `EnqueueRenderAction` method, which can be used to explicitly dispatch actions to the compositor thread. The method returns a `Task`, with or without a return value, representing an action scheduled for execution on the compositor thread during the next render pass.

The returned task can be awaited from the UI thread, allowing subsequent code to wait until the queued action has completed. This makes it possible to safely perform operations that require access to the VTK rendering context in response to events that originate on the UI thread.

For example, an action such as hardware picking can be queued in response to a UI event, such as a button click, without directly accessing the rendering context from the UI thread:

```CSharp
private async void OnClick(object? sender, RoutedEventArgs e)
{
  RenderWindowControl? ctrl = this.FindControl<RenderWindowControl>("VTKControl");
  vtkRenderer ren = ctrl.RenderWindow.GetRenderers().GetFirstRenderer();
  Task<int> pickTask = ctrl.EnqueueRenderAction(() =>
  {
    vtkPropPicker picker = vtkPropPicker.New();
    return picker.Pick(300, 300, 0, ren);
  });
  // Request render to run the task
  ctrl.RequestNextFrameRendering();
  // Block execution until picking is done
```

```
  int picked = await pickTask;
  Console.WriteLine("Picked async : " + picked);
}
```

Conversely, to schedule work from the compositor thread back to the UI thread, use Avalonia's built-in dispatching mechanisms, such as `Dispatcher.UIThread.Post`.

💡 The `RenderWindowControl`'s render window is locked while rendering is in progress. When accessing or manipulating VTK objects from multiple threads, synchronize access to the render window to avoid concurrent operations that could interfere with rendering.
When appropriate, this can be done by acquiring the render window's lock before performing the operation:

```CSharp
lock (RenderWindow)
{
    // Manage concurrent operations on VTK objects here
}
```

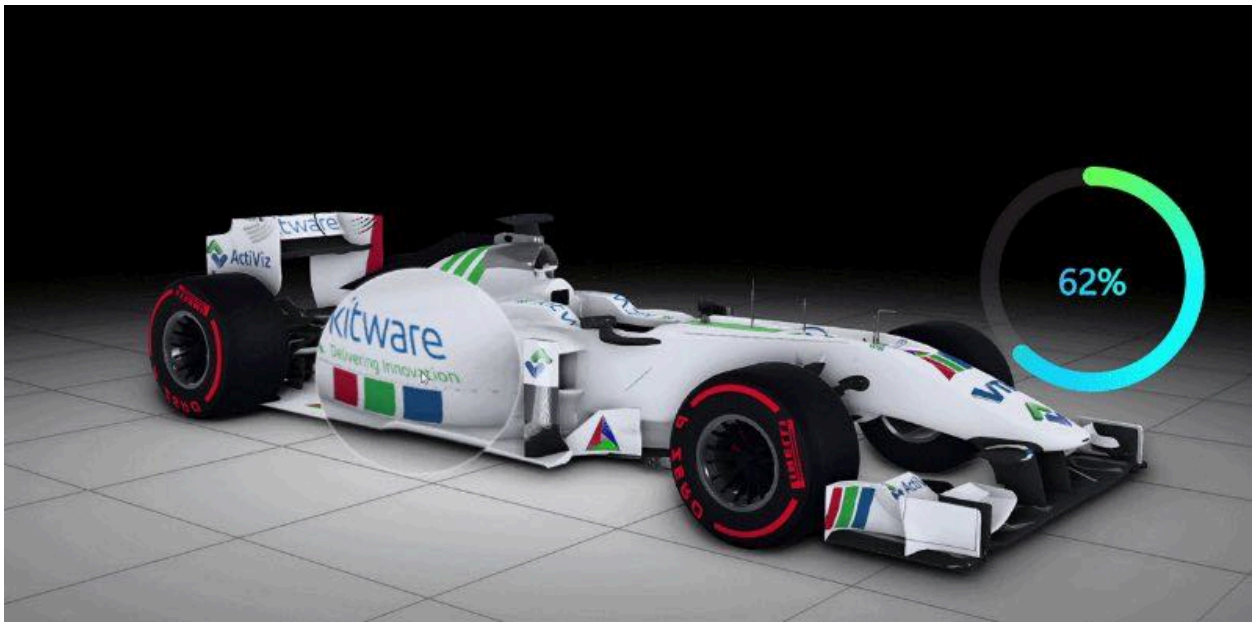
❖ Interlacing VTK and SkiaSharp rendering

SkiaSharp is a cross-platform 2D graphics API used by Avalonia to render UI components. The `RenderWindowControl` class leverages offscreen rendering of the VTK scene into the Skia canvas presented by Avalonia. This approach enables VTK and Skia rendering commands to be seamlessly interleaved, making it possible to combine 3D visualization with custom 2D graphics to create advanced visual effects such as magnifying glasses, glassmorphism overlays, or animated progress gauges.

To simplify the integration of VTK and SkiaSharp, the `RenderWindowControl` exposes two `Action<SKCanvas>` delegates that allow applications to issue Skia rendering commands immediately before and after the VTK rendering pass:

```CSharp
// PreRender action called before VTK rendering pass
// e.g. Clear background before rendering VTK
renderWindowControl.PreRender = (SKCanvas canvas) => {
```

```
  canvas.Clear(SKColors.DarkGray)
};

// PostRender action called after VTK rendering pass.
// e.g. Render a blurred version of VTK's framebuffer
renderWindowControl.PostRender = (SKCanvas canvas) => {
  using var snapshot = canvas.Surface?.Snapshot();
  using var blurPaint = new SKPaint
  {
    ImageFilter = SKImageFilter.CreateBlur(12, 12)
  };
  canvas.DrawImage(snapshot, 0, 0, blurPaint);
};
```



Example of magnifying glass and progress gauge implemented with Skia in PostRender actions

5. Core concepts and Best practices

5.1. Objects lifetime

5.1.1. Constructor

While VTK objects can be instantiated using the standard C# `new` operator, prefer VTK's static `New()` method, especially for classes subject to VTK's factory mechanism. Using `New()` ensures that the C# wrapper reflects the actual VTK object type created by the factory.

For example, `var actor = new vtkActor()` creates a C# object of type `vtkActor`, even though the underlying C++ object may be a `vtkOpenGLActor` as a result of VTK's factory mechanism. Although `vtkOpenGLActor.SafeDownCast(actor)` can retrieve the appropriate wrapper, object comparisons may fail because the two C# objects have different types. In particular, when `actor` is created with `new`, `actor` is not equal to `vtkOpenGLActor.SafeDownCast(actor)`.

Using `new` also prevents the expected use of the C# `as` operator:
`var actor = new vtkActor(); actor as vtkOpenGLActor;` returns null.
In contrast, when using VTK's factory method:
`var actor = vtkActor.New(); actor as vtkOpenGLActor;` the cast works as expected.

Finally, using `New()` follows the standard VTK idiom and makes the code immediately recognizable to developers familiar with VTK.

5.1.2. Dispose and Finalizer

ActiViz manages the lifetime of the underlying C++ vtkObject instances by incrementing and decrementing their reference counts as required.

When a managed C# VTK object becomes unreachable, its finalizer is eventually executed by the garbage collector. The finalizer internally calls `Dispose()` on the VTK wrapper, which decrements the reference count of the underlying native object and clears the native reference held by the managed C# object. Consequently, it is not necessary to explicitly call `Dispose()` on every VTK object: objects that are no longer referenced will eventually be finalized and their native references released automatically.

Such finalization is non-deterministic, `Dispose()` can therefore be used when deterministic resource release is required, for example to release native memory at a specific point during execution. When called, `Dispose()` invokes `GC.SuppressFinalize()`, preventing the finalizer from running later and thus avoiding a second release or decrement of the native

object's reference count.

Releasing a wrapper's native reference does not necessarily destroy the underlying C++ `vtkObject` immediately, as it may still be referenced by other native VTK objects. Once all native references have been released and the object's VTK reference count reaches zero, VTK automatically destroys and frees the underlying native object.

5.2. VTK events and observers

ActiViz supports VTK's object event and observer mechanism through a higher-level .NET API based on C# delegates and events. Observers can be added and removed using the following pattern:

```CSharp
_object.<EventName>Evt += callback; // Add an observer
_object.<EventName>Evt -= callback; // Remove an observer
```

`EventName` corresponds to one of the VTK `vtkCommand` event names. To avoid naming conflicts, the `Event` suffix used by VTK is replaced with `Evt`. For example, to observe `ModifiedEvent`, use the `ModifiedEvt` event handler.

All ActiViz event delegates use the following signature:

```CSharp
void Handler(vtkObject sender, vtkObjectEventArgs e)
```

The `sender` parameter identifies the VTK object that raised the event. If necessary, you can cast it back to its specific type using `SafeDownCast`:

```
var renderer = vtkRenderer.SafeDownCast(sender);
```

This is particularly useful when the same callback is registered with events from multiple VTK objects.

The `e` parameter provides access to optional event data when such data is supplied by the event. Refer to the VTK `vtkCommand` documentation for details about the data associated with specific events.

You can also register observers using lambda expressions:

```CSharp
_object.ModifiedEvt += (sender, e) =>
{
    // Handle the event
};
```

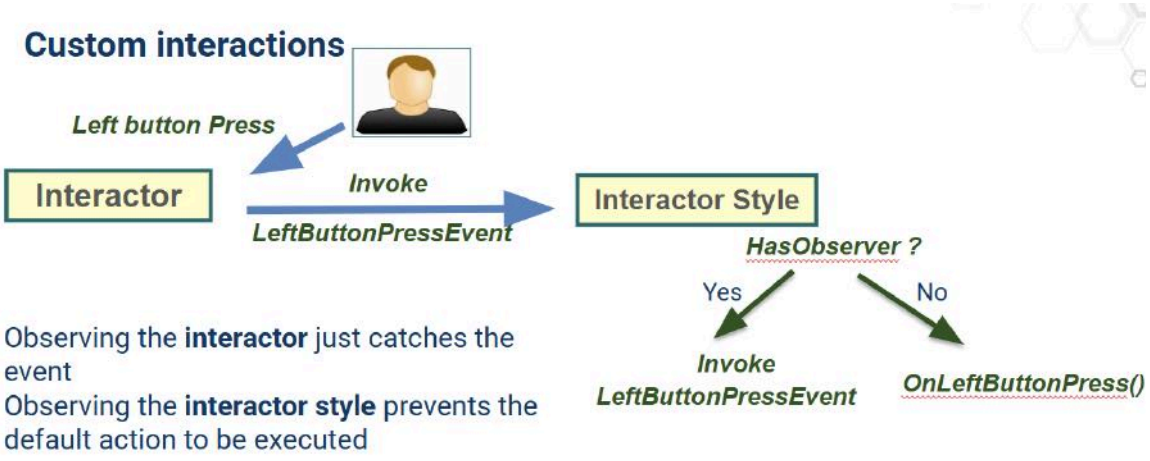
⚠️**WARNING:** vtkCommand and `vtkCallbackCommand` shall not be used to observe VTK events in ActiViz as they require subclassing VTK objects or passing function pointers as arguments, which is not fully supported.

❖ Keyboard and Mouse Events

Keyboard and mouse interactions can be observed through either a `vtkRenderWindowInteractor` or a `vtkInteractorStyle` instance.

Adding an observer to a `vtkRenderWindowInteractor` provides passive event observation: your callback receives the event, but the interactor's normal processing continues.

Adding an observer to a `vtkInteractorStyle`, on the other hand, replaces the style's default handler for the corresponding interaction event. This allows you to customize or intercept the default behavior.
If you want to execute custom code before allowing the interactor style's normal behavior to continue, add an observer to the interactor style and then explicitly invoke the corresponding `On<Event>` method.



Example:

CSharp

```
var interactor = vtkRenderWindowInteractor.New();
interactor.MouseMoveEvt += (vtkObject sender, vtkObjectEventArgs e) => {
   Console.WriteLine("Mouse move"); // Do something meaningful
};

var style = vtkInteractorStyleTrackballCamera.New();
interactor.SetInteractorStyle(style);
style.CharEvt += (vtkObject sender, vtkObjectEventArgs e) => {
  var style = vtkInteractorStyle.SafeDownCast(sender)
              ?? throw new Exception("Wrong type");

  if(style.GetInteractor().GetKeySym() == "p")
  {
    Console.WriteLine("P has been pressed!"); // Do something meaningful
  }
  else
  {
    style.OnChar(); // keep default behavior for other cases
  }
};
```

6. Supported Platforms

ActiViz is available across a range of operating systems, processor architectures, and UI frameworks. Because support differs depending on the platform and technology combination, compatibility is grouped into tiers. These classifications are reviewed with each ActiViz release and may change over time as platform support evolves.

✅Tier 1 — Fully Supported

Platforms in this tier are officially supported, tested, and included with the ActiViz license.

🟦Tier 2 — Supported Through Commercial Engagement

Platforms in this tier are not currently part of the standard ActiViz distribution and do not receive full official product support. However, their technical viability has been demonstrated through proof-of-concept implementations or previous customer projects.

Support for these platforms may be provided through a limited commercial support engagement, subject to technical assessment and scope. They may also be planned or prioritized for future development and could be promoted to Tier 1 in a future ActiViz release.

❌Tier 3 — Exploratory

Platforms in this tier are not currently supported and are not included in the ActiViz development roadmap, or their technical feasibility has not yet been established.

Support for these platforms may require substantial investigation and advanced engineering work undertaken within the scope of a commercial support engagement.

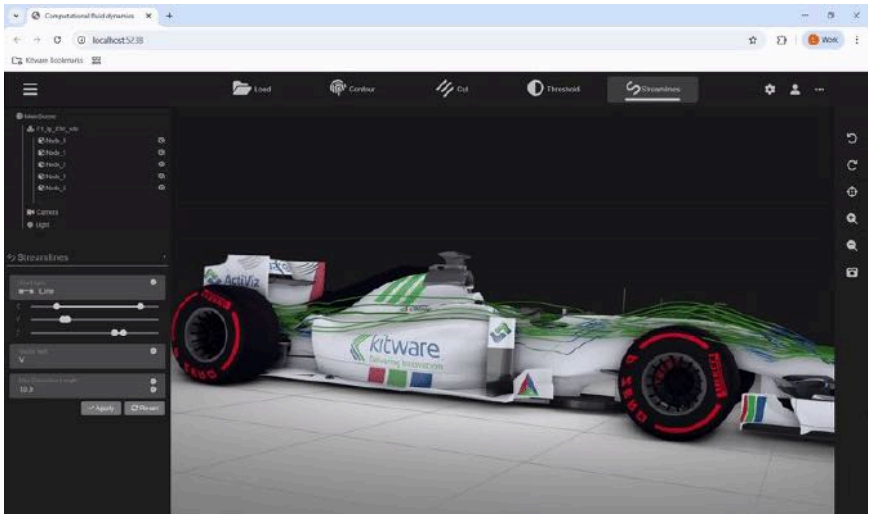
The table below covers only the platforms and frameworks that have been evaluated or considered so far. It is not intended to be exhaustive or to represent a definitive statement of current or future compatibility. If you are interested in integrating ActiViz into an environment that is not listed, please contact us.

	Windows		Linux		MacOS		Web	Mobiles
	x64	ARM64	x64	ARM64 (*)	x64 (**)	ARM64	WASM	iOS / Android
Console app	✅Tier 1	🟦Tier 2	✅Tier 1	🟦Tier 2	🟦Tier 2	✅Tier 1	✅Tier 1	❌Tier 3
Winforms	✅Tier 1	–	–	–	–	–	–	–
WPF	✅Tier 1	–	–	–	–	–	–	–
WinUI	✅Tier 1	–	–	–	–	–	–	–
Avalonia	✅Tier 1	🟦Tier 2	✅Tier 1	🟦Tier 2	🟦Tier 2	✅Tier 1	🟦Tier 2	❌Tier 3
Uno Platform	🟦Tier 2	–	🟦Tier 2	–	🟦Tier 2	–	❌Tier 3	❌Tier 3
Unity 2022 (***)	✅Tier 1	🟦Tier 2	❌Tier 3	❌Tier 3	❌Tier 3	❌Tier 3	❌Tier 3	❌Tier 3
GTK	🟦Tier 2	🟦Tier 2	🟦Tier 2	🟦Tier 2	❌Tier 3	❌Tier 3	–	–
Blazor (****)	–	–	–	–	–	–	🟦Tier 2	–
.NET MAUI (****)	❌Tier 3	❌Tier 3	–	–	❌Tier 3	❌Tier 3	–	❌Tier 3

(*) : Including embedded platforms
(**) : Deprecated in ActiViz 9.5
(***) : Support for recent versions is considered Tier 3
(****): Integration on desktop possible with specific WebView components

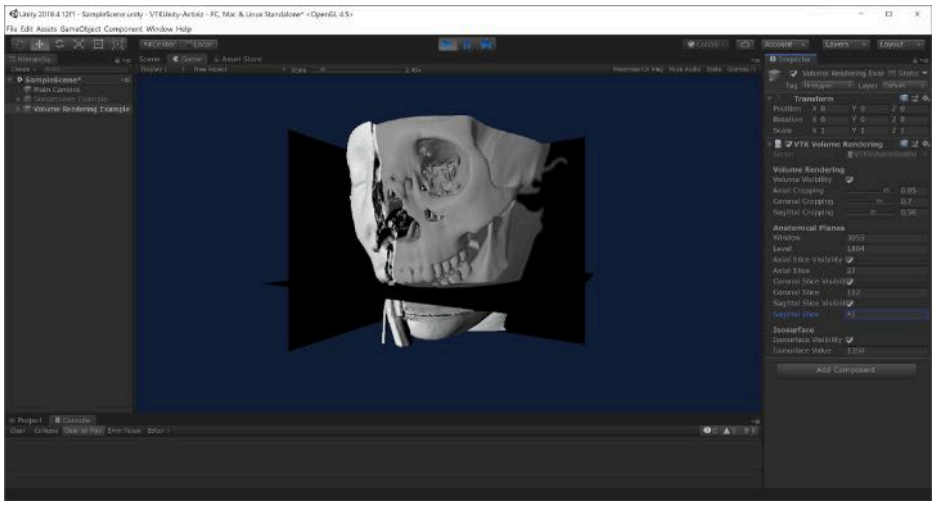
7. ActiViz on the Web

Starting with ActiViz 9.5, it is now possible to use ActiViz .NET in your web browser, thanks to VTK WASM support, mono-wasm and Emscripten project.
To start a WASM project, please take a look at the HelloVTK-WASM example `README.md` file.



8. ActiViz in Unity

ActiViz enables the integration of the high-performance algorithms, data representation, scalability, and visualization techniques offered by VTK into the high quality and convenient environment that Unity provides.
Check out Kitware's blog for more information or contact us to request a trial version.



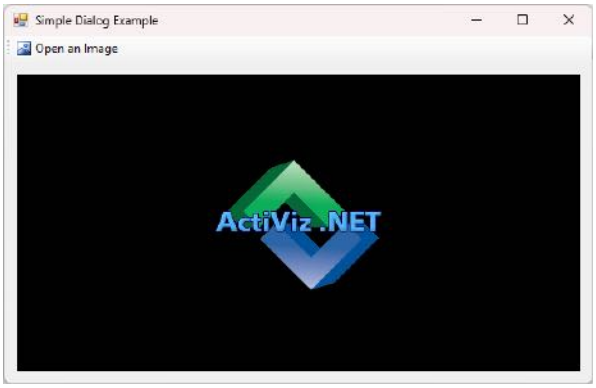
9. Examples

In the following examples, we dispense with the details of the code and project creation. Rather we provide an eclectic mix of examples demonstrating some of the power of VTK, including highlighting some code snippets that are relevant to key functionality. The complete code, in C# and Visual Basic, is available in the installation directory under the Examples subdirectory.

Important Notes: These examples were written to be simple, clear demonstrations of the potential of ActiViz .NET. In general less than an hour was spent writing these examples, and even the Wikipedia browser was completed in well under a day. Thus Kitware does not claim that these are bulletproof applications meant for industrial application. Further, there are some specific limitations of which you should be aware:

- If you install the software in the "Program Files" directory (or other privileged location), then you will have to build the software with admin privileges, or preferably, copy the examples to a non-privileged location and fix the appropriate reference paths.
- The two example applications, the Wikipedia Browser and the File Browser, may experience problems when requests lead to processing large amounts of data. Please refer to the specific examples for further clarification.

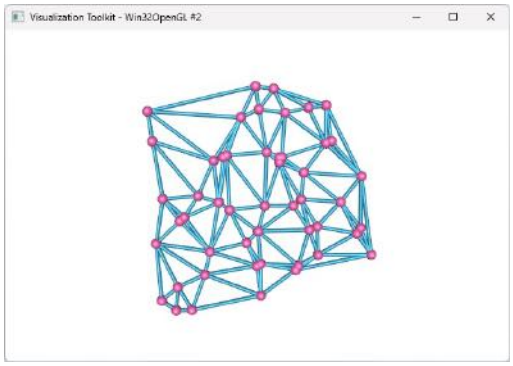
9.1. Load Image Files Dialog



This example demonstrates a simple dialog to open and display an image. It supports several file formats including .png, .jpg, .jpeg, .tif, .slc, .dicom, .minc, .bmp, and .pmn. It will also support .vtk files, which supports data types including polygonal data, structured grids, and unstructured grids.

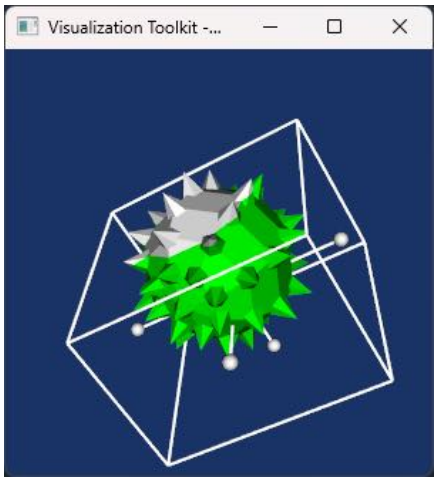
This image viewer is unusual in that it places the image on a quadrilateral polygon using texture mapping. The polygon exists in 3D space so it is possible to move the camera around the polygon using the render window interactor. The "Simple Dialog Example" illustrates what the application looks like after loading a simple image file.

9.2. Delaunay Triangulation



The Delaunay triangulation is a construct in computational geometry used to generate triangulations (i.e., polygonal meshes where the polygons are all triangles). It has many useful functions including interpolating data. In this example, a random set of points in a 2D plane is generated. Next, VTK filters are used to place tubes around the edges and ball glyphs at the mesh vertices. You may wish to play with the number of points generated, as well as the appearance of the graph by modifying the underlying C# source code.

9.3. Box Widget



VTK has an extensive set of widgets. Widgets are objects that appear in the scene but may be directly interacted with. Widgets are analogous to 2D GUI devices such as buttons and sliders, except in 3D they can take on much more complex forms. The figure to the right shows one very general widget called the box widget.

The box widget has six faces that can be selected and then rotated. Each face can be translated separately to modify the extent of the box. The box can also be uniformly scaled and translated. The widget can be queried to return useful information such as the current transformation matrix, and the six planes that form the box. In VTK, this information can be used by the data processing pipeline to perform additional operations. For example in the figure shown, the six planes are used to clip the mace. The surface of the mace outside of the box is grey, the surface inside is green. Also, when you run the example, keypress—i is used to enabled/disable the widget (it will appear and disappear from the scene).

One of the important features of this example is the use of events and associated callbacks to couple the widget with other VTK objects. In this example, we define a callback function as follows:

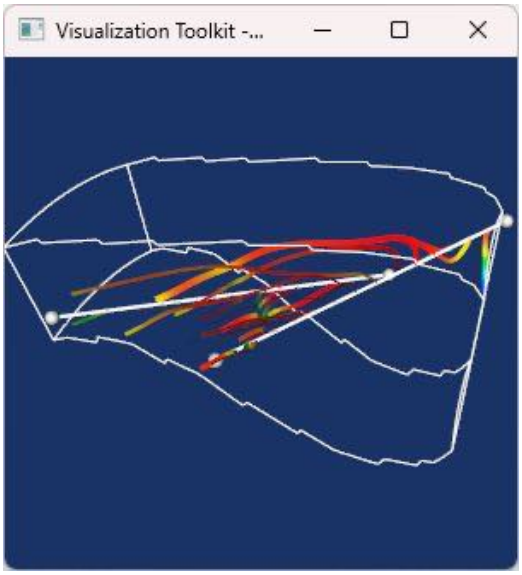
```
public static void
    SelectPolygons(vtkObject sender, vtkObjectEventArgs e)
{
    boxWidget.GetPlanes((vtkPlanes)planes);
    selectActor.VisibilityOn();
}
```

Here the “planes” object is a collection of the six planes retrieved from the widget. In turn, the planes define a clip function to the vtkClipPolyData filter. Next, the callback function is connected to the box widget by observing the end interaction event. (Typically widgets provide three events: start interaction, interaction, and end interaction.)

```
boxWidget.EndInteractionEvt +=
     new vtkObject.vtkObjectEventHandler(SelectPolygons);
```

Thus, when the widget is manipulated, the event is triggered (at the end of motion) and the callback function is invoked. Thus widgets are very easy to add and to use in ActiViz.

9.4. Streamline Generation



This example shows some of the visualization capabilities of VTK. A structured grid (think of a volume warped in 3D space so that it is topologically regular but geometrically distorted) is read with associated scalar and vector data. This 3D dataset is the result of a computational simulation of combustion in a segment of an annular combustor from an aircraft engine.

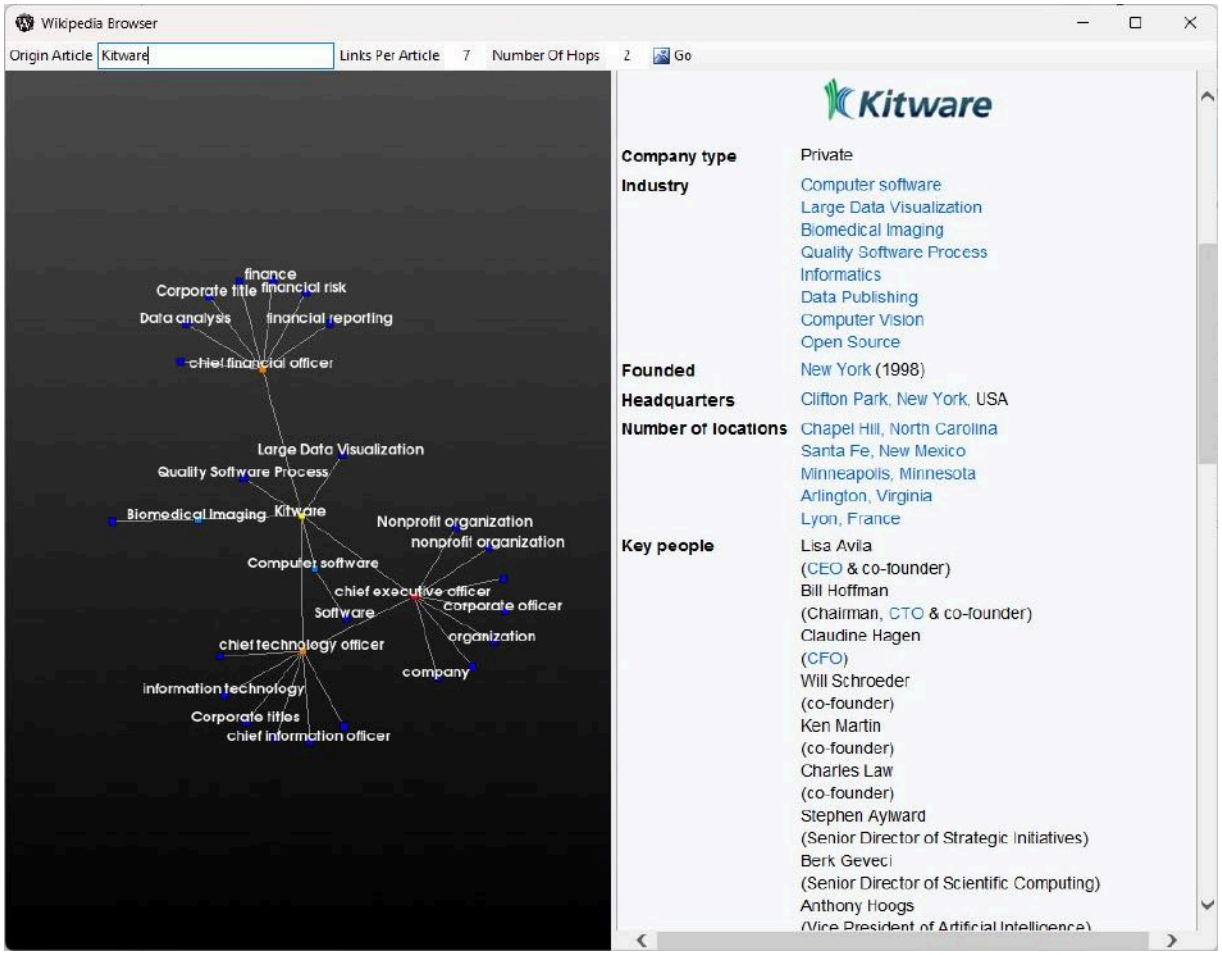
To visualize the data, streamlines are generated. These synthetic streamlines are similar to smoke traces used in wind tunnels, and represent the path that a massless particle would take in a vector field (here the vector field is the flow momentum). The streamlines are modified by using a VTK filter to place a ribbon on the streamline. Since streamlines must have a starting point, a line widget is used to seed the streamlines.

Once you compile and run the example, aside from using the interactor to position the camera, it is possible to move the line widget. Do this by selecting the widget end points (marked as little balls), or grabbing the line and translating it around. It is also possible to modify the code to increase the line resolution in order to generate more streamlines emanating from it.

9.5. Wikipedia Browser

The general idea of this application is to enable browsing of Wikipedia (the free encyclopedia wikipedia.org) pages. The Wikipedia Browser example uses VTK's information visualization classes to show relationships between Wikipedia entries as a graph (or network). You can interact with this graph to see how articles relate to one another.

The application consists of two panels: on the left a graph browser indicating the relationship of Wikipedia articles to one another; on the right, the Wikipedia web page corresponding to the recently selected graph node (see Figure below).



The application enables the user to type in a search string as a start point for the browser (make sure to select the "Go" button to initiate the search). In the figure below, the initial search string is "VTK". Given this initial string, the search expands out into *N* links where *N* is specified by the user (here the default value is 10). The first *N* links in the initial string are

followed to other Wikipedia pages, and this process continues *H* times, where *H* is the user specified number of hops to follow. Note that *N* and *H* must be carefully specified. If these numbers are big enough, the graph may grow rapidly (it is possible to crash the application as currently written in *N* and *H* are too large). Further, since the nodes represent a web page, each page must be accessed over the internet, and then parsed, which can be very slow depending on network performance.

Once the initial graph is specified, it is possible to expand it iteratively. Simply use the left mouse button to select a rectangular selection region in the left panel. Any nodes in the selection region are further expanded in the graph. Note that when expanded, VTK's graph layout algorithm will run and can reposition the existing graph nodes. You may also select a single node by left-mouse clicking on the node. If you select a single node, its corresponding Wikipedia page is shown in the right panel. If you select a group of nodes with a rectangular selection, then an arbitrary selection from the group is made and the corresponding Wikipedia page is shown in the right panel.

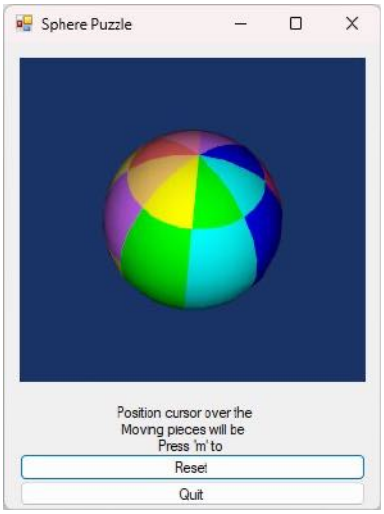
The graph layout has some features that facilitate navigation. While the left mouse button is used for selection, the middle mouse button can be used to translate the graph, and the right mouse button to zoom in and out (move the mouse "up" to zoom, and "down" to zoom out). Zooming in and out causes text to appear and disappear dynamically. Finally, if your mouse has a scrolling wheel, scrolling the wheel also zooms in and out on the graph.

The source code for this application is included in the ActiViz .NET examples directory; feel free to extend it. Other VTK classes exist to improve the behavior of this application, including ways to adjust graph layout, control mouse bindings, change the selection process, and populate the scene.

9.6. Sphere Puzzle

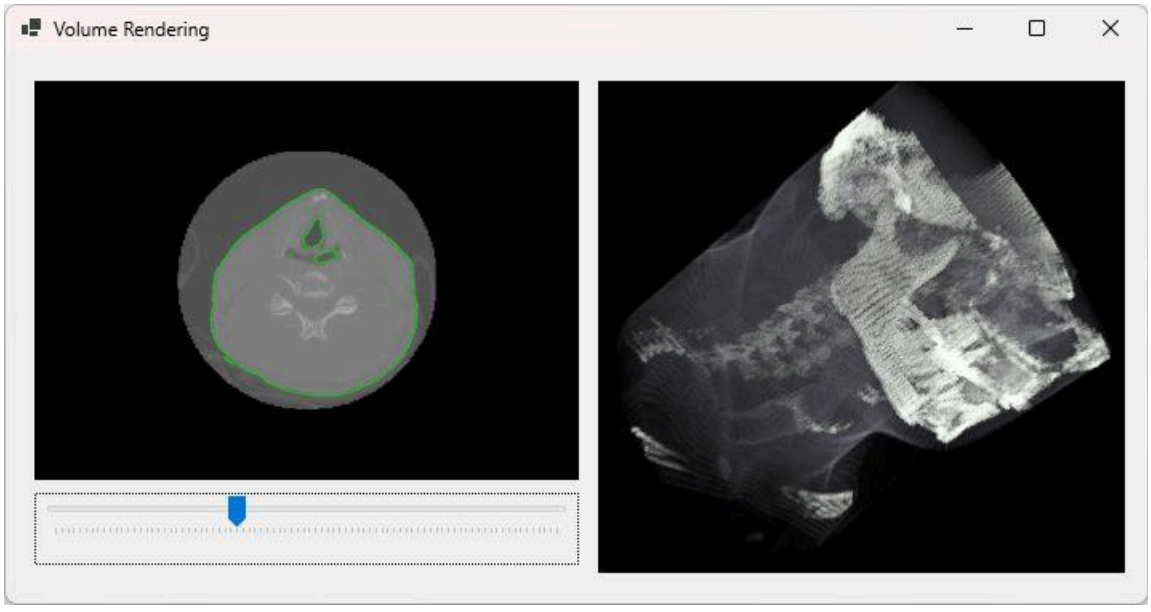
This is a cute example much like a Rubik's cube puzzle except on a sphere rather than a cube. After the application initializes itself by randomizing the panels on the sphere, the user attempts to restore the sphere to the proper coloring (to see the final coloring, hit the "Reset" button).

Moving the sphere panels involves rotating the sphere either in the longitudinal or latitudinal directions. By moving the mouse pointer close to a latitude or longitude line, a portion of the sphere will light up, indicating the portion that will rotate. By hitting the "m" key, the highlighted portion of the sphere will rotate. Repeat this process until you return the sphere to the proper coloring.



9.7. Volume Rendering

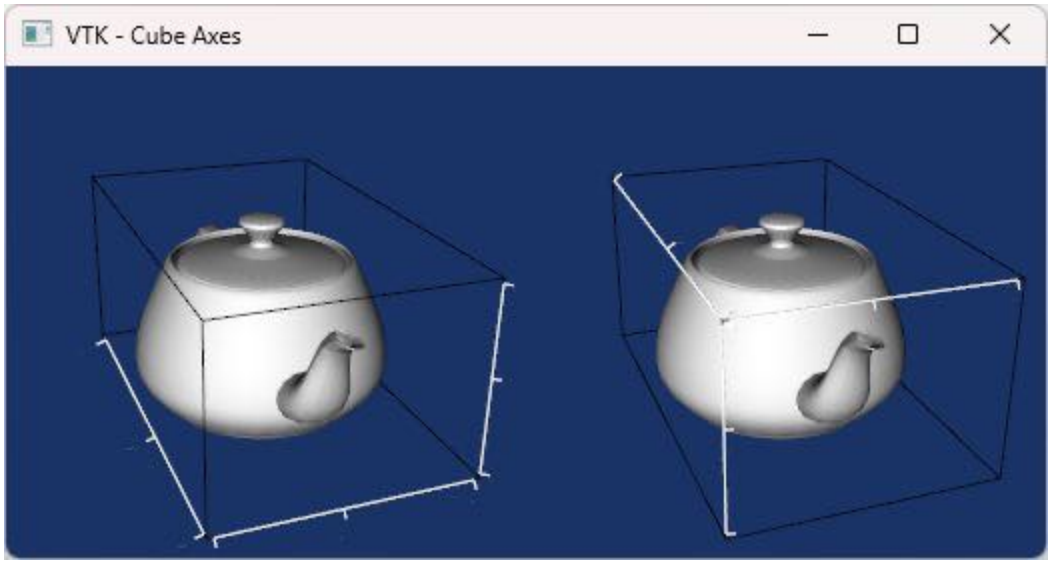
Volume rendering is a visualization technique for rendering regularly sampled 3D data (think of a stack of 2D images or slices, with a uniform vertical spacing between slices). While there are many ways of implementing volume rendering, probably the simplest way to think about it as a ray casting algorithm. Imagine that the volume varies in transparency, where part of the volume may be fully transparent, and part opaque. Further the volume may vary in color as well. A transfer function controls the transparency and color; the transfer function typically maps the volume data value (e.g., intensity) into a color and transparency value. Then to volume render, rays are cast from the camera through each pixel in the renderer. Some may pass into and then traverse the volume, while other rays may miss the volume entirely. For those rays that intersect the volume, sample points are selected, in order, along the ray and mapped through the transfer function based on the data values at each sample point. The color and transparency is accumulated until the ray exits the volume, or the pixel becomes fully opaque. In the past, volume rendering was a compute intensive, typically slow process. However, with modern CPUs and GPUs, volume rendering performance realizes interactive frame rates. VTK implements several different volume rendering strategies as implemented by vtkVolumeMapper and subclasses.



In this example, two render windows are used. One window is connected to a slider and shows the volume slices. The second window shows a volume-rendered image. The second window supports the standard interactor bindings. On modern computers, the application will be fully interactive. (Note: careful examination of the volume rendering window shows that level-of-detail (LOD) is being used while interacting with the data. Once the mouse is released, the rendering reverts to full resolution. LOD strategies are common in computer graphics and VTK supports this in a variety of ways. See vtkLODActor, for example.)

9.8. Cube Axes Actor

VTK provides many classes for annotating data. In the following example, two renderers are placed in a single render window, and a vtkCubeAxesActor is used to annotate the data as shown in the figure below. The two instances of vtkCubeAxesActor are configured differently so that one always emanates from the corner of the object's bounding box closest to the camera, and the other follows the closest edges of the bounding box. Also, a camera is shared between the two renderers so that the two views are synchronized with each other.



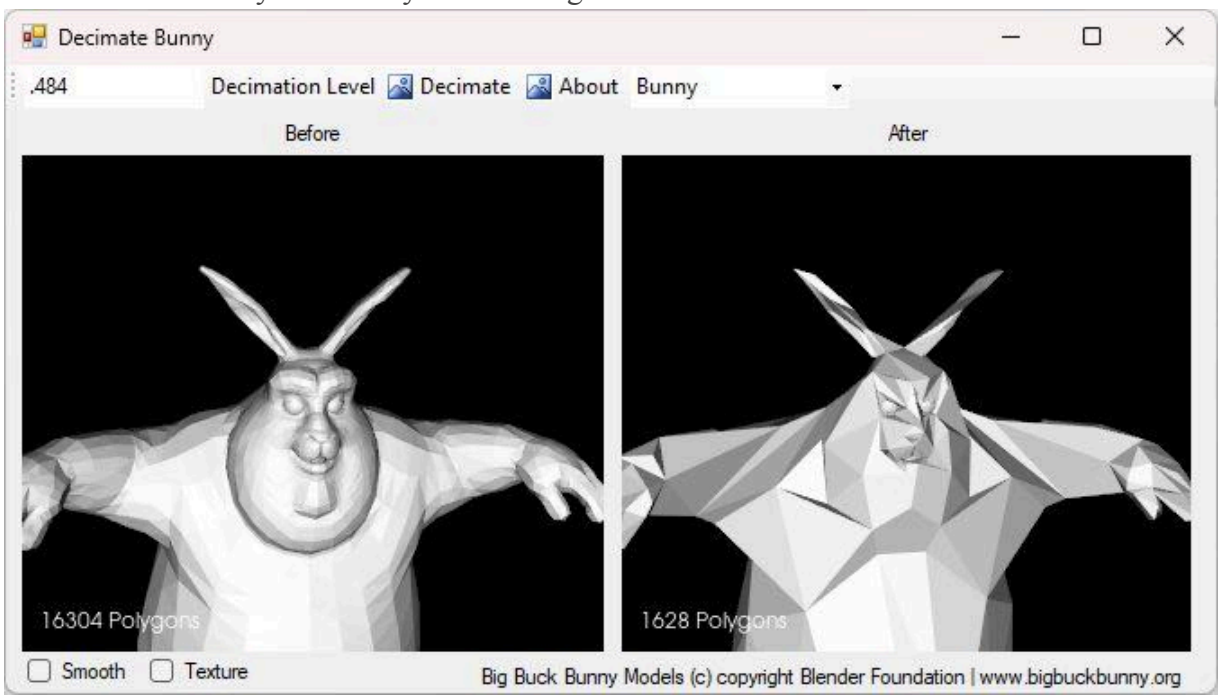
Other types of annotation in VTK include text (both 2D, residing on top of the 3D geometry, or 3D, embedded in the 3D scene); captions, popup balloons, axes, labels, *x*-*y* plots, and many other classes.

9.9. Decimation

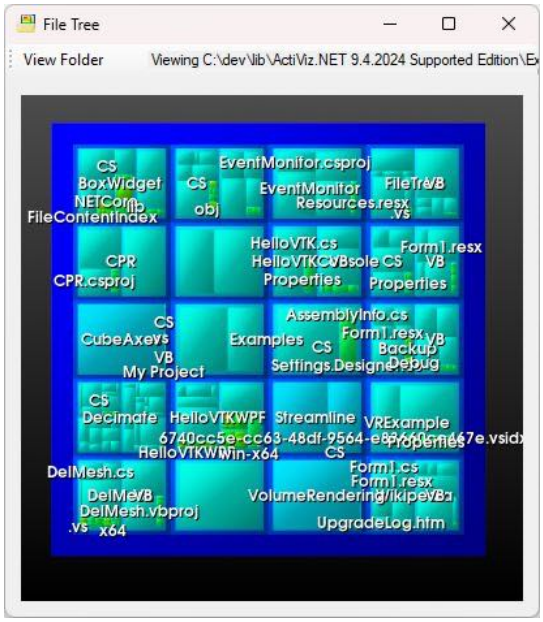
Decimation, or polygon reduction, is a technique used in computer graphics to reduce the complex of geometric models. For example, in the figure below, the original model of *Big Buck Bunny* contains over 16,000 polygons; the decimated model contains approximately 8,000 polygons. In computer graphics and visualization, it is common to produce models with large polygon counts. Such models can be difficult to manipulate due to the delays in rendering and processing the data. Decimation is used to limit data sizes and thereby improve interaction rates.

In this example, models from the open-source movie *Big Buck Bunny* are used to demonstrate decimation (models copyright the Blender Foundation | www.bigbuckbunny.org). VTK provides several different decimation algorithms with varying levels of speed and fidelity. The fastest algorithm is vtkQuadricClustering, the algorithm that generates models with the best fidelity is vtkQuadricDecimation. In the example above, vtkDecimatePro is used. (See

the included documentation for more information.) Since the source code is provided, interested readers may want to try different algorithms.



9.10. File Browser



The next example is a useful application for quickly locating large files and directories in a directory tree. The application uses VTK's tree map class and other information visualization classes. A tree map is a layout scheme that represents the "size" of each node in a tree (including the node's children) by a rectangle. In turn, the children are embedded in the parent's rectangle in a recursive process. The meaning of size varies depending on the application. In this example, size means the disk space usage of a file (if a leaf node) or the disk usage of the files and subdirectories contained within it (if an intermediate, directory node).

To use the application, simply load the directory you are interested in. Depending on the size of the directory and the subdirectories and files contained within it, the program may take anywhere from one or two seconds to minutes to execute. Once the image (shown) is generated, you can interact

with the display. Labels are placed in the center of each rectangle and dynamically resize based as the user zooms in and out towards the tree map. Further, by moving the mouse pointer over the tree map a label will appear indicating the current directory or file at which you are pointing. Selecting (left mouse button) brings up a file browser. Use the right mouse button to zoom, and the center button to pan across the tree map.

(A cautionary note: if you run this example from Visual Studio in debug mode in very large directories, timeouts may occur that produce errors. The solution is not to run in debug mode.)

10. For More Information

The following section lists additional resources to learn more about VTK and Kitware.

10.1. Manual Pages

The ActiViz .NET product is distributed with extensive code documentation on a per class basis. Refer to this documentation for details regarding the use of the examples shown here. A good way to view this is through Visual Studio's object browser view. Choose "Object Browser" from the view menu, and then browse through the Kitware.VTK classes.

10.2. VTK.org Web Site

The VTK open-source community maintains resources at http://www.vtk.org. Besides on-line documentation, user's and developer's mailing lists are available for posting and receiving answers to questions. Refer to vtk.org for more information on joining mailing lists.

10.3. More Examples

VTK has hundreds of additional examples available from which to learn about it. These examples are written in C++ or other VTK wrapping languages such as Python. In most cases these examples may be directly translated into a .NET supported programming language. A large number of tests, used to ensure the quality of VTK, are also useful references for learning about VTK. These tests and examples are great jumping off points for ActiViz users who wish to write their own .NET applications.

VTK examples: https://examples.vtk.org/site/Cxx

VTK tests are available in VTK source code, most tests are located in `<kit>/<module>/Testing/Cxx` for example https://gitlab.kitware.com/vtk/vtk/-/tree/master/Rendering/Core/Testing/Cxx.

10.4. VTK Books

VTK was created in 1993 as part of a textbook published by Prentice-Hall. Because the software was useful, and because of its open-source license, a community quickly grew up around the system. Since that time, the textbook has evolved and a supplemental *VTK User's*

Guide has been written. These two books, listed below, are available through Kitware at http://www.kitware.com/products/books.html and Amazon.com.

- *The Visualization Toolkit An Object-Oriented Approach to 3D Graphics*. Schroeder, Martin, Lorensen — this is principally a theory book on visualization, although it contains many practical examples.
- *The VTK User's Guide* — shows how to use VTK through an extensive set of examples.

10.5. Related Software

Kitware creates and distributes many open-source and proprietary software systems. A synopsis of some of these systems follows.

Open Source Software (BSD or BSD-style licenses)

- The Visualization Toolkit (VTK) — provides 3D visualization capabilities including information visualization, volume rendering, modeling, data processing, and human-computer interaction tools (http://www.vtk.org).
- The Parallel Visualization Application (ParaView) — built on VTK, ParaView is a distributed, scalable visualization application designed for data sizes ranging from small to very large (http://www.paraview.org).
- The Insight Segmentation and Registration Toolkit (ITK) — provides a comprehensive suite of image processing, registration and segmentation tools for biomedical and other imaging tasks (http://www.itk.org).
- CMake, CTest, CPack, and CDash — these systems form the core of Kitware's quality software process. CMake is used to manage cross-platform development. CTest and CDash are the software testing client and server, respectively. CPack is used to package and distribute software across multiple computing platforms (http://www.cmake.org, http://www.cdash.org).
- 3D Slicer — is a biomedical application built on VTK and ITK. It has been successfully employed for medical and biological imaging applications (http://www.slicer.org).

ActiViz .NET OpenSource Edition

ActiViz .NET Supported Edition