Table of Contents

Getting Started

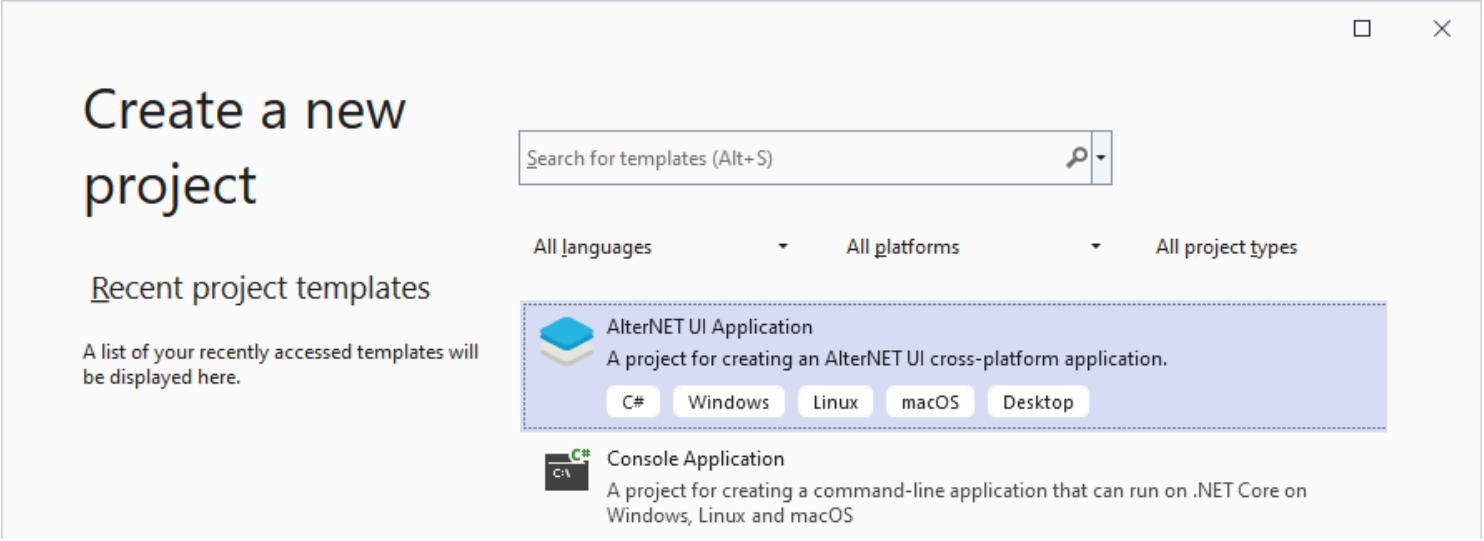
AlterNET UI allows you to develop light-footprint cross-platform .NET desktop applications.

You can use your favorite .NET development environments on Windows, macOS, and Linux to develop AlterNET UI applications: Microsoft Visual Studio and Visual Studio Code.

IDE Support

Microsoft Visual Studio

To use AlterNET UI with Visual Studio, you need to install AlterNET UI extension for Visual Studio. This extension adds a new project type - **AlterNET UI Application** and a new project item type - **UIXML file**.



Refer to our Visual Studio step-by-step tutorial for more details.

Visual Studio Code

You will need to use a command-line tool to download AlterNET UI project template and create AlterNET UI application.

Refer to our command line step-by-step tutorial for more details.

AlterNET UI is published on NuGet.org as a package you can use in your .NET projects.

Building Blocks

The basic building blocks of a typical AlterNET UI application are uncomplicated. One such class is Application, which allows to start and stop an application, and a Window, which represents an on-screen window to display UI elements inside it. A UI inside a Window is usually defined by a pair of the UIXML markup code file and C# (**code-behind**) file with event handlers and programming logic.

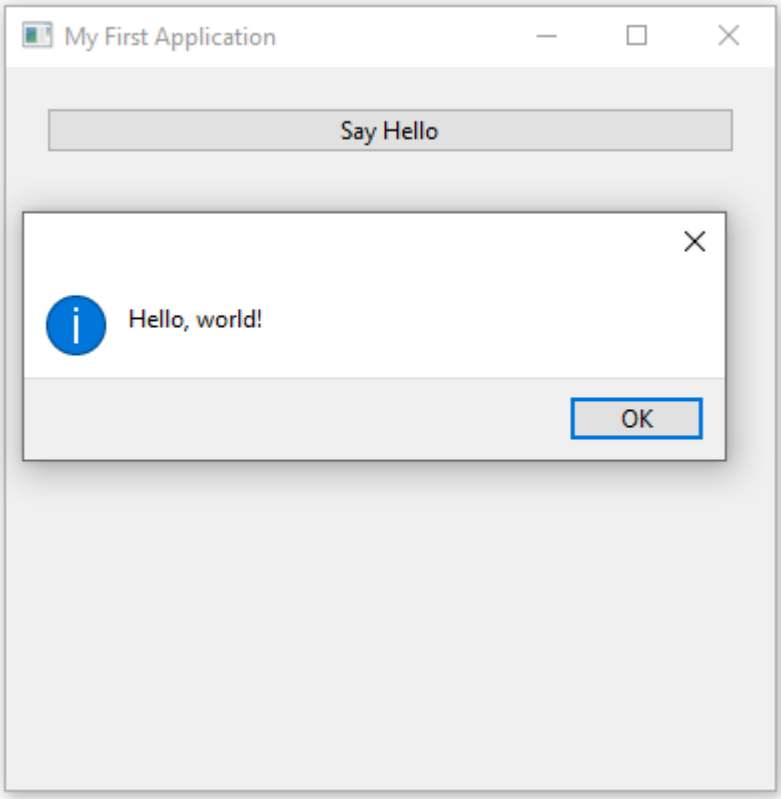
UIXML markup code is very similar to XAML. Using UIXML follows an approach of separating visual layout from code. The visual layout is then defined by a declarative UIXML document and the code-behind files, which are written in C#. The application logic is implemented in these code-behind files. This approach is inspired by WPF design and is proven by widespread industry use.

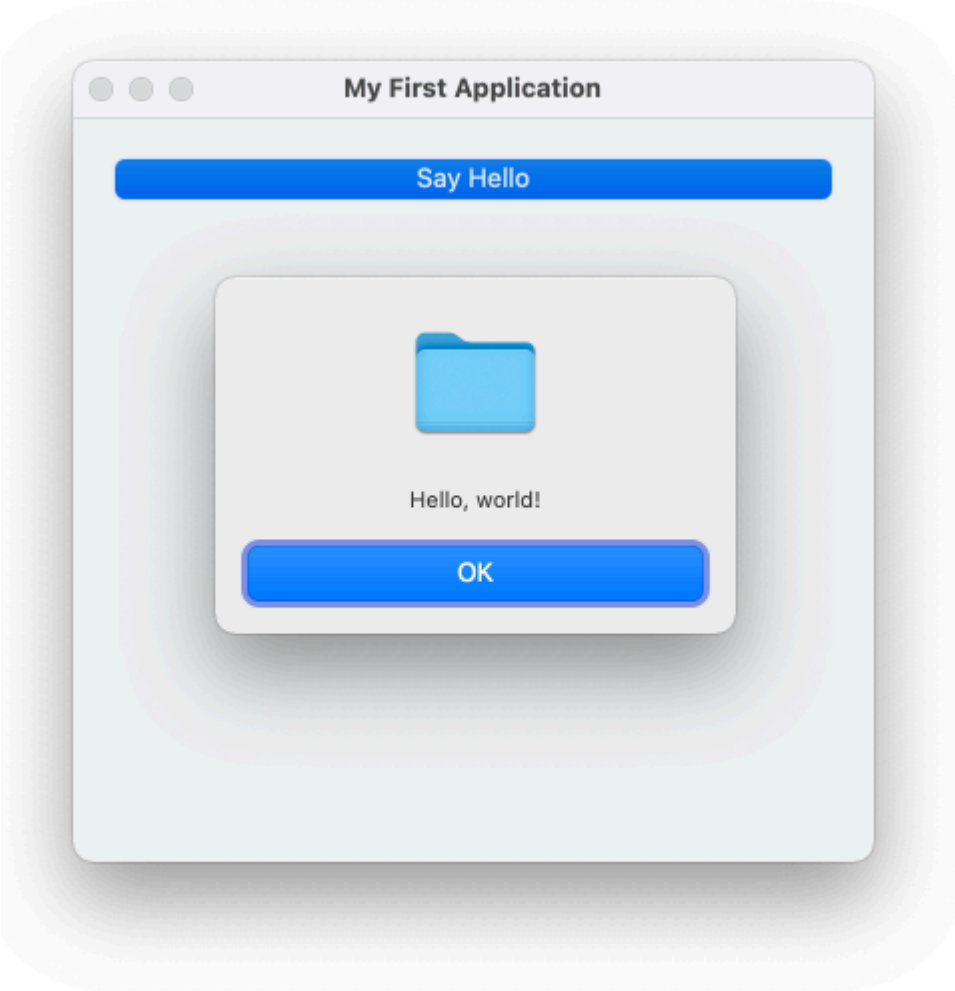
The following example shows how you can create a window with a few controls as part of a user interface.

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20" Click="HelloButton_Click" />
  </StackPanel>
</Window>
```

```
using System;
using Alternet.UI;

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

        private void HelloButton_Click(object? sender, EventArgs e)
        {
            MessageBox.Show("Hello, world!");
        }
    }
}
```

Here is how this application looks on different operating systems:


My First Application
Say Hello
Hello, world!
OK


My First Application
Say Hello
Hello, world!
OK

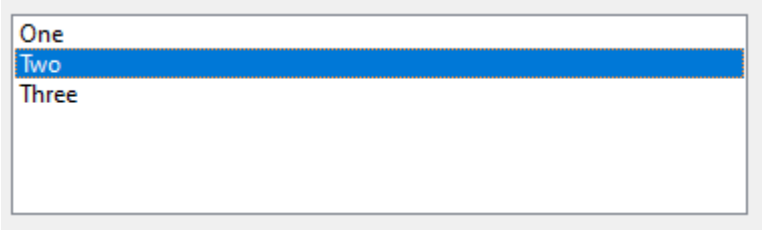
Explore more examples on GitHub.

Controls

AlterNET UI provides a set of standard controls which use native API and look and feel precisely like native elements on all platforms and different screen resolutions.

Examples of how a ListBox can look on different platforms:

Windows | macOS | Linux



AlterNET UI provides the following core controls:

Containers: Grid, StackPanel, VerticalStackPanel, HorizontalStackPanel, GroupBox, Border, TabControl, SplittedPanel, LayoutPanel.

These controls act as containers for other controls and provide a different kinds of layouts in your windows.

A ScrollViewer is a special kind of container which makes its child controls scrollable.

Inputs controls: Button, CheckBox, ComboBox, RadioButton, NumericUpDown, TextBox, DateTimePicker and Slider.

These controls most often detect and respond to user input. The control classes expose API to handle text and mouse input, focus management, and more.

Data display: ListBox, ListView, TreeView, CheckListBox, VirtualListBox, PropertyGrid.

These controls provide a visual representation of data elements in different layouts or views.

Html display: WebBrowser.

This control may be used to render full featured web documents.

Informational: Label, ProgressBar.

These controls are designed to present information to the user in a visual form.

In AlterNET UI, each control is defined within a rectangle that represents its boundaries. The actual size of this rectangle is calculated by the layout system at runtime using automatic measurements based on the available screen size, parent properties, and element properties such as border, width, height, margin, and padding.

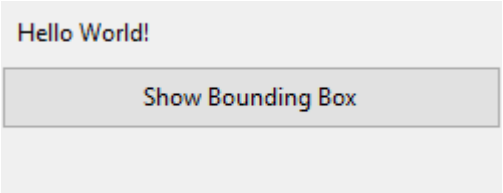
Layout

This topic describes the AlterNET UI layout system. Understanding how and when layout calculations occur is essential for creating user interfaces in AlterNET UI.

Control Bounding Boxes

When thinking about layout in AlterNET UI, it is important to understand the bounding box that surrounds all controls. Each Control consumed by the layout system can be thought of as a rectangle that is slotted into the layout. The size of the rectangle is determined by calculating the available screen space, the size of any constraints, layout-specific properties (such as margin and padding), and the individual behavior of the parent control. By processing this data, the layout system can calculate the position of all the children of a particular Control. It is important to remember that sizing characteristics, defined on the parent control, such as a Border, affect its children.

The following illustration shows a simple layout.

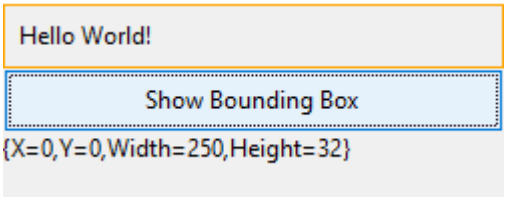


This layout can be achieved by using the following UIXML.

```
<Grid Name="myGrid" Height="150">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Border Name="border1" Grid.Column="0" Grid.Row="0" BorderBrush="{x:Null}">
        <Label Margin="5" Text="Hello World!" />
    </Border>
    <Button Width="125" Height="25" Grid.Column="0" Grid.Row="1" Text="Show Bounding Box"
Click="ShowBoundingBoxButton_Click" />
    <Label Name="txt2" Grid.Column="1" Grid.Row="2"/>
</Grid>
```

A Label control is hosted within a Grid. While the text fills only the upper-left corner of the first column, the allocated space for the containing Border is actually much larger. The bounding box of any Control

can be retrieved by using the Bounds method. The following illustration shows the bounding box for the Label control.



As shown by the orange rectangle, the allocated space for the Label control is actually much larger than it appears. As additional controls are added to the Grid, this allocation could shrink or expand, depending on the type and size of controls that are added.

The layout bounds of the Border are highlighted by setting the BorderColor property.

```
private void ShowBoundingBoxButton_Click(object? sender, EventArgs e)
{
    border1.BorderBrush = Brushes.Orange;
    txt2.Text = border1.Bounds.ToString();
}
```

The Layout System

At its simplest, the layout is a recursive system that leads to control being sized, positioned, and drawn. More specifically, layout describes the process of measuring and arranging the members of a Control's Children collection. The layout is an intensive process. The larger the Children collection, the greater the number of calculations that must be made. Complexity can also be introduced based on the layout behavior defined by the Control control that owns the collection. A relatively simple layout Control, such as Border, can have significantly better performance than a more complex Control, such as Grid.

Each time that a child Control changes its position, it has the potential to trigger a new pass by the layout system. Therefore, it is important to understand the events that can invoke the layout system, as unnecessary invocation can lead to poor application performance. The following describes the process that occurs when the layout system is invoked.

1. A child Control generally begins the layout process by first measuring itself by having its core sizing properties evaluated, such as Width, SuggestedWidth, SuggestedHeight, Height, and Margin.

2. After that, a custom GetPreferredSize implementation may change the desired control's size.

3. Layout using Dock property.

4. Layout Control-specific logic is applied, such as StackPanel's OnLayout logic and its related properties, such as Orientation.

5. The control bounds are set after all children have been measured and laid out.

6. The process is invoked again if additional Children are added to the collection, or the PerformLayout method is called.

Measuring and Positioning Children

The layout system typically performs two operations for each member of the Children collection, a measure and a layout. Each child Control provides its own GetPreferredSize and OnLayout methods to achieve its own specific layout behavior.

By default, a control provides a base measure and layout logic. It considers several base control inputs to perform its operation.

First, native size properties of the Control are evaluated, such as Visible. Secondly, the properties which affect the value of the control's preferred size are processed. These properties generally describe the sizing characteristics of the underlying Control, such as its Height, Width, Margin, Padding, Layout, Dock, HorizontalAlignment, and VerticalAlignment. Each of these properties can change the space that is necessary to display the control.

The ultimate goal of the measurement process is for the child to determine its preferred size, which occurs during the GetPreferredSize call.

During the layout process, the parent Control control generates a rectangle that represents the bounds of the child. This value is set to the Bounds property.

The layout logic evaluates the preferred size of the child and evaluates any additional properties that may affect the actual size of the control, such as margin and alignment, and puts the child within its layout slot. The child does not have to (and frequently does not) fill the entire allocated space. After that, the layout process is complete.

Standard Layout Controls

AlterNET UI includes a group of controls that enable complex layouts. For example, stacking controls can easily be achieved by using the StackPanel control, while more complex layouts are possible by using a Grid.

The following table summarizes the available layout controls.

Control name	Description
Grid	Defines a flexible grid area that consists of columns and rows.

Control name	Description
StackPanel	Arranges child controls into a single line that can be oriented horizontally or vertically.
VerticalStackPanel	Arranges child controls into a single line that can be oriented vertically.
HorizontalStack Panel	Arranges child controls into a single line that can be oriented horizontally.
SplittedPanel	Manages subcontrols which are aligned to the sides with splitter control between them.
LayoutPanel	Arranges child controls using different methods.
Splitter	Provides resizing of docked controls.

Custom Layout Behaviors

For applications that require a layout that is not possible by using any of the predefined controls, custom layout behaviors can be achieved using one of these approaches:

- Set Layout property.
- Inherit from Control and override the GetPreferredSize and OnLayout methods.
- Implement CustomLayout event handler.
- Implement GlobalOnLayout and/or GlobalGetPreferredSize event handlers.

Input Overview

This article explains the architecture of the input systems in AlteNET UI.

Input API

The primary input API exposure is found on the base element classes: UIElement, FrameworkElement and Control. These classes provide functionality for input events related to key presses, mouse buttons, mouse-wheel, mouse movement, focus management, and mouse capture, to name a few. By placing the input API on the base elements, rather than treating all input events as a service, the input architecture enables the input events to be sourced by a particular object in the UI and to support an event routing scheme whereby more than one element has an opportunity to handle an input event.

Keyboard and Mouse Classes

In addition to the input API on the base element classes, the Keyboard class and Mouse classes provide additional API for working with keyboard and mouse input.

Examples of input API on the Keyboard class are the Modifiers property, which returns the ModifierKeys currently pressed, and the IsKeyDown method, which determines whether a specified key is pressed.

The following example uses the GetKeyStates method to determine if a Key is in the down state.

```
// Uses the Keyboard.GetKeyStates to determine if a key is down.
// A bitwise AND operation is used in the comparison.
// e is an instance of KeyEventArgs.
if ((Keyboard.GetKeyStates(Key.Enter) & KeyStates.Down) > 0)
{
    btnNone.Background = Brushes.Red;
}
```

An example of input API on the Mouse class is MiddleButton, which obtains the state of the middle mouse button.

The following example determines whether the LeftButton on the mouse is in the Pressed state.

```
if (Mouse.LeftButton == MouseButtonState.Pressed)
{
    UpdateSampleResults("Left Button Pressed");
}
```

The Mouse and Keyboard classes are covered in more detail throughout this overview.

Event Routing

A FrameworkElement can contain other elements as child elements in its content model, forming a tree of elements. In AlterNET UI, the parent element can participate in input directed to its child elements or other descendants by handing events. This is especially useful for building controls out of smaller controls, a process known as "control composition" or "compositing."

Event routing is the process of forwarding events to multiple elements so that a particular object or element along the route can choose to offer a significant response (through handling) to an event that might have been sourced by a different element.

Routed events use one of three routing mechanisms: direct, bubbling, and tunneling. In direct routing, the source element is the only element notified, and the event is not routed to any other elements. However, the direct routed event still offers some additional capabilities that are only present for routed events as opposed to standard CLR events. Bubbling works up the element tree by first notifying the element that sourced the event, then the parent element, and so on. Tunneling starts at the root of the element tree and works down, ending with the original source element.

Handling Input Events

To handle an element's input, an event handler must be associated with that particular event. In UIXML this is straightforward: you reference the name of the event as an attribute of the element that will be listening for this event. Then, you set the value of the attribute to the name of the event handler that you define, based on a delegate. The event handler must be written in code such as C# and can be included in a code-behind file.

Keyboard events occur when the operating system reports key actions that occur while the keyboard focus is on an element. Mouse and stylus events each fall into two categories: events that report changes in pointer position relative to the element and events that report changes in the state of device buttons.

Keyboard Input Event Example

The following example listens for a left arrow key press. A StackPanel is created that has a Button. An event handler to listen for the left arrow key press is attached to the Button instance.

The first section of the example creates the StackPanel and the Button and attaches the event handler for the KeyDown.

```
<StackPanel>
    <Button Background="AliceBlue"
            KeyDown="OnButtonKeyDown"
            Text="Button1"/>
</StackPanel>
```

```
// Create the UI elements.
StackPanel keyboardStackPanel = new StackPanel();
Button keyboardButton1 = new Button();

// Set properties on Buttons.
keyboardButton1.Background = Brushes.AliceBlue;
keyboardButton1.Text = "Button 1";

// Attach Buttons to StackPanel.
keyboardStackPanel.Children.Add(keyboardButton1);

// Attach event handler.
keyboardButton1.KeyDown += new KeyEventHandler(OnButtonKeyDown);
```

The second section is written in code and defines the event handler. When the left arrow key is pressed, and the Button has keyboard focus, the handler runs and the Background color of the Button is changed. If the key is pressed, but it is not the left arrow key, the Background color of the Button is changed back to its starting color.

```
private void OnButtonKeyDown(object sender, KeyEventArgs e)
{
    Button source = e.Source as Button;
    if (source != null)
    {
        if (e.Key == Key.Left)
        {
            source.Background = Brushes.LemonChiffon;
        }
        else
        {
            source.Background = Brushes.AliceBlue;
        }
    }
}
```

Mouse Input Event Example

In the following example, the Background color of a Button is changed when the mouse pointer enters the Button. The Background color is restored when the mouse leaves the Button.

The first section of the example creates the StackPanel and the Button control and attaches the event handlers for the MouseEnter and MouseLeave events to the Button.

```
<StackPanel>
    <Button Background="AliceBlue"
            MouseEnter="OnMouseExampleMouseEnter"
            MouseLeave="OnMosueExampleMouseLeave"
            Text="Button">
    </Button>
</StackPanel>
```

```
// Create the UI elements.
StackPanel mouseMoveStackPanel = new StackPanel();
Button mouseMoveButton = new Button();

// Set properties on Button.
mouseMoveButton.Background = Brushes.AliceBlue;
mouseMoveButton.Text = "Button";

// Attach Buttons to StackPanel.
mouseMoveStackPanel.Children.Add(mouseMoveButton);

// Attach event handler.
mouseMoveButton.MouseEnter += new MouseEventHandler(OnMouseExampleMouseEnter);
mouseMoveButton.MouseLeave += new MouseEventHandler(OnMosueExampleMouseLeave);
```

The second section of the example is written in code and defines the event handlers. When the mouse enters the Button, the Background color of the Button is changed to SlateGray. When the mouse leaves the Button, the Background color of the Button is changed back to AliceBlue.

```
private void OnMouseExampleMouseEnter(object sender, MouseEventArgs e)
{
    // Cast the source of the event to a Button.
    Button source = e.Source as Button;

    // If source is a Button.
    if (source != null)
    {
        source.Background = Brushes.SlateGray;
    }
}
```

```
private void OnMosueExampleMouseLeave(object sender, MouseEventArgs e)
{
    // Cast the source of the event to a Button.
```

```
    Button source = e.Source as Button;

    // If source is a Button.
    if (source != null)
    {
        source.Background = Brushes.AliceBlue;
    }
}
```

Text Input

The KeyPress event enables you to listen for text input in a device-independent manner. The keyboard is the primary means of text input, but speech, handwriting, and other input devices can generate text input also.

For keyboard input, AlterNET UI first sends the appropriate KeyDown/KeyUp events. If those events are not handled, and the key is textual (rather than a control key such as directional arrows or function keys), then a KeyPress event is raised. There is not always a simple one-to-one mapping between KeyDown/KeyUp and KeyPress events because multiple keystrokes can generate a single character of text input, and single keystrokes can generate multi-character strings. This is especially true for languages such as Chinese, Japanese, and Korean, which use Input Method Editors (IMEs) to generate the thousands of possible characters in their corresponding alphabets.

The following example defines a handler for the Click event and a handler for the KeyDown event.

The first segment of code or markup creates the user interface.

```
<StackPanel KeyDown="OnTextInputKeyDown">
    <Button Click="OnTextInputButtonClick" Text="Open" />
    <TextBox />
</StackPanel>
```

```
// Create the UI elements.
StackPanel textInputStackPanel = new StackPanel();
Button textInputButton = new Button();
TextBox textInputTextBox = new TextBox();
textInputButton.Text = "Open";

// Attach elements to StackPanel.
textInputStackPanel.Children.Add(textInputeButton);
textInputStackPanel.Children.Add(textInputTextBox);

// Attach event handlers.
```

```
textInputStackPanel.KeyDown += new KeyEventHandler(OnTextInputKeyDown);
textInputButton.Click += new RoutedEventHandler(OnTextInputButtonClick);
```

The second segment of the code contains the event handlers.

```
private void OnTextInputKeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.O && Keyboard.Modifiers == ModifierKeys.Control)
    {
        Handle();
        e.Handled = true;
    }
}

private void OnTextInputButtonClick(object sender, EventArgs e)
{
    Handle();
    e.Handled = true;
}

public void Handle()
{
    MessageBox.Show("Pretend this opens a file");
}
```

Because input events bubble up the event route, the StackPanel receives the input regardless of which element has keyboard focus. The TextBox control is notified first, and the `OnTextInputKeyDown` handler is called only if the TextBox did not handle the input.

Mouse Position

The AlterNET UI input API provides helpful information with regard to coordinate spaces. For example, coordinate `(0,0)` is the upper-left coordinate, but the upper-left of which element in the tree? The element that is the input target? The element you attached your event handler to? Or something else? To avoid confusion, the AlterNET UI input API requires that you specify your frame of reference when you work with coordinates obtained through the mouse. The GetPosition method returns the coordinate of the mouse pointer relative to the specified element.

Mouse Capture

Mouse devices specifically hold a modal characteristic known as mouse capture. Mouse capture is used to maintain a transitional input state when a drag-and-drop operation is started so that other operations involving the nominal on-screen position of the mouse pointer do not necessarily occur. During the

drag, the user cannot click without aborting the drag-and-drop, which makes most mouseover cues inappropriate while the mouse capture is held by the drag origin. The input system exposes APIs that can determine the mouse capture state, as well as APIs that can force mouse capture to a specific element or clear the mouse capture state.

The Input System and Base Elements

Input events such as the attached events defined by the Mouse and Keyboard classes are raised by the input system and injected into a particular position in the object model based on hit testing the visual tree at run time.

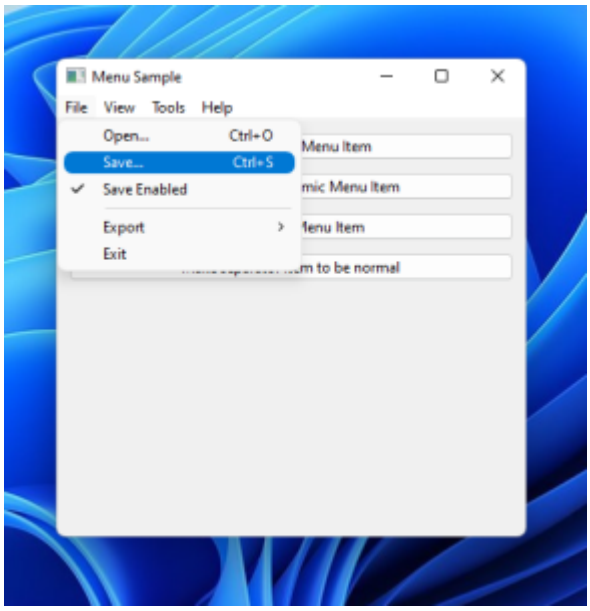
Each of the events that Mouse and Keyboard define as an attached event is also re-exposed by the base element class UIElement as a new routed event. The base element routed events are generated by classes handling the original attached event and reusing the event data.

When the input event becomes associated with a particular source element through its base element input event implementation, it can be routed through the remainder of an event route that is based on a combination of logical and visual tree objects, and be handled by application code. Generally, it is more convenient to handle these device-related input events using the routed events on UIElement, because you can use more intuitive event handler syntax both in UIXML and in code. You could choose to handle the attached event that initiated the process instead, but you would face several issues: the attached event may be marked handled by the base element class handling, and you need to use the accessor methods rather than true event syntax in order to attach handlers for attached events.

Menu

AlterNET UI allows building menus on all the platforms it supports; see the screenshots below.

Windows | macOS | Linux



The Menu and MainMenu classes enable you to organize elements associated with commands and event handlers in a hierarchical order. Each Menu element contains a collection of MenuItem elements.

Menu Control

The Menu control presents a list of items that specify commands or options for an application. Typically, clicking a MenuItem opens a submenu or causes an application to execute a command.

Creating Menus

The following example creates a MainMenu with Menu items inside. The Menu contains MenuItem objects that use the Command, Text, Checked properties and the Click event.

```
<Window.Menu>
  <MainMenu>
    <MenuItem Text="_File">
      <MenuItem Text="_Open..." Name="openMenuItem" Click="OpenMenuItem_Click"
Shortcut="Ctrl+O"/>
      <MenuItem Text="_Save..." Name="saveMenuItem" Command="{Binding SaveCommand}"/>
      <MenuItem Text="-" Name="separatorMenuItem" />
      <MenuItem Text="E_xit" Name="exitMenuItem" Click="ExitMenuItem_Click" />
    </MenuItem>
```

```
    <MenuItem Text="_View">
      <MenuItem Text="_Grid" Checked="True" Name="gridMenuItem" Click="GridMenuItem_Click"/>
    </MenuItem>
  </MainMenu>
</Window.Menu>


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

        DataContext = this;
        SaveCommand = new Command(o => MessageBox.Show("Save"));
    }

    public Command SaveCommand { get; }

    private void OpenMenuItem_Click(object sender, EventArgs e) => MessageBox.Show("Open");

    private void GridMenuItem_Click(object sender, EventArgs e) => MessageBox.Show("Grid
item is checked: " + gridMenuItem.Checked);

    private void ExitMenuItem_Click(object sender, EventArgs e) => Close();
}
```

MenuItems with Keyboard Shortcuts

Keyboard shortcuts are character combinations that can be entered with the keyboard to invoke Menu commands. For example, the shortcut for **Copy** is CTRL+C. To assign a keyboard shortcut to a menu item, use the Shortcut property.

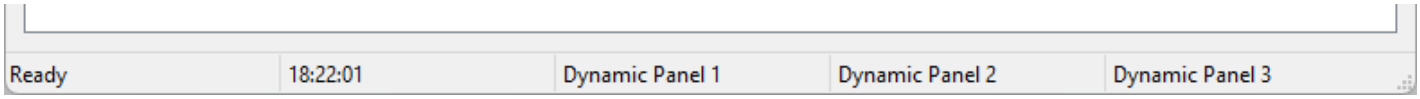
MenuItem roles for macOS application menu support

On macOS, by the system UI guidelines, the **About**, **Quit**, and **Preferences** items must be placed into the application (the leftmost) menu. They also should have standard names and keyboard shortcuts. On Windows and Linux, these items are usually located in different menus, like **Help**, **File**, and **Tools**. As many applications include these menu items, AlterNET UI provides automatic role-based menu item location adjustment on macOS. Usually, the developer does not have to do anything, as the framework automatically deduces item roles from the menu item names and relocates them to the required menu on macOS. For cases when more fine control is required, please use Role property, and MenuItemRoles class.

Using StatusBar

The AlterNET UI StatusBar control is used as an area, usually displayed at the bottom of a window, where an application can display various status information. StatusBar controls can have status bar panels that display text to indicate the state of the open document; for example, that the document is modified.

Windows | macOS | Linux



Using the StatusBar Control

Internet Explorer uses a status bar to indicate the URL of a page when the mouse rolls over the hyperlink; Microsoft Word gives you information on page location, section location, and editing modes such as overtype and revision tracking; and Visual Studio uses the status bar to provide context-sensitive information, such as telling you how to manipulate dockable windows as either docked or floating.

A status bar is divided into panels to display information using the Panels property. The StatusBar control allows you to create status bar panels by adding StatusBarPanel objects to a Panels collection. Each StatusBarPanel object should have Text assigned to be displayed in the status bar.

Working with the StatusBar Control

The following example shows how to use StatusBar component:

```
<Window>
    <Window.StatusBar>
        <StatusBar Name="statusBar">
            <StatusBarPanel Text="Ready" />
            <StatusBarPanel Name="clockStatusBarPanel" />
        </StatusBar>
    </Window.StatusBar>
</Window>
```

```
Timer clockTimer;

public MainWindow()
{
    InitializeComponent();
```

```
    clockTimer = new Timer(TimeSpan.FromMilliseconds(200), (o, e) =>
clockStatusBarPanel.Text = DateTime.Now.ToString("HH:mm:ss"));
    clockTimer.Start();
}
```

Drag-and-Drop Support

You can enable user drag-and-drop operations within an AlterNET UI application by handling a series of events, most notably the DragEnter, DragLeave, and DragDrop events.

Drag-and-drop events

There are two categories of events in a drag-and-drop operation: events that occur on the current target of the drag-and-drop operation and events that occur on the source of the drag-and-drop operation. To perform drag-and-drop operations, you must handle these events. By working with the information available in the event arguments of these events, you can easily facilitate drag-and-drop operations.

Events on the current drop target

The following table shows the events that occur on the current target of a drag-and-drop operation.

Mouse Event	**Description**
DragEnter	This event occurs when an object is dragged into the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragOver	This event occurs when an object is dragged while the mouse pointer is within the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragDrop	This event occurs when a drag-and-drop operation is completed. The handler for this event receives an argument of type DragEventArgs.
DragLeave	This event occurs when an object is dragged out of the control's bounds. The handler for this event receives an argument of type EventArgs.

The DragEventArgs class provides the location of the mouse pointer, the current state of the mouse buttons and modifier keys of the keyboard, the data being dragged, and DragDropEffects values that specify the operations allowed by the source of the drag event and the target drop effect for the operation.

Performing drag-and-drop

Drag-and-drop operations always involve two components, the **drag source** and the **drop target**. To start a drag-and-drop operation, designate a control as the source and handle the MouseDown event. In the event handler, call the DoDragDrop method providing the data associated with the drop and the a DragDropEffects value.

Set the target control's AllowDrop property set to `true` to allow that control to accept a drag-and-drop operation. The target handles two events. First, an event in response to the drag being over the control, such as DragOver. And a second event which is the drop action itself, DragDrop.

The following example demonstrates a drag from a Label control to a TextBox. When the drag is completed, the `TextBox` responds by assigning the label's text to itself.

```
textBox1.AllowDrop = true;
// ...

// Initiate the drag
private void label1_MouseDown(object sender, MouseEventArgs e) =>
    DoDragDrop(((Label)sender).Text, DragDropEffects.Copy);

// Set the effect filter and allow the drop on this control
private void textBox1_DragOver(object sender, DragEventArgs e) =>
    e.Effect = DragDropEffects.Copy;

// React to the drop on this control
private void textBox1_DragDrop(object sender, DragEventArgs e) =>
    textBox1.Text = (string)e.Data.GetData(DataFormats.Text);
```

Dragging Data

All drag-and-drop operations begin with dragging. The functionality to enable data to be collected when dragging begins is implemented in the DoDragDrop method.

In the following example, the MouseDown event is used to start the drag operation because it is the most intuitive (most drag-and-drop actions begin with the mouse button being pressed). However, remember that any event could be used to initiate a drag-and-drop procedure.

To start a drag operation

1. In the MouseDown event for the control where the drag will begin use the `DoDragDrop` method to set the data to be dragged and the allowed effect dragging will have. For more information, see Data and Effect.

The following example shows how to initiate a drag operation. The control where the drag begins is a Button control, the data being dragged is the string representing the Text property of the Button control, and the allowed effects are either copying or moving.

```
private void button1_MouseDown(object sender, Alternet.UI.MouseEventArgs e)
{
```

```
    button1.DoDragDrop(button1.Text, DragDropEffects.Copy |
        DragDropEffects.Move);
}
```

Dropping Data

Once you have begun dragging data from a location on a window or control, you will naturally want to drop it somewhere. The cursor will change when it crosses an area of a window or control that is correctly configured for dropping data. Any area within a window or control can be made to accept dropped data by setting the AllowDrop property and handling the DragEnter and DragDrop events.

To perform a drop

1. Set the AllowDrop property to true.

2. In the `DragEnter` event for the control where the drop will occur, ensure that the data being dragged is of an acceptable type (in this case, Text). The code then sets the effect that will happen when the drop occurs to a value in the DragDropEffects enumeration. For more information, see Effect.

```
private void textBox1_DragEnter(object sender, Alternet.UI.DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.Text))
        e.Effect = DragDropEffects.Copy;
    else
        e.Effect = DragDropEffects.None;
}
```

3. In the DragDrop event for the control where the drop will occur, use the GetData method to retrieve the data being dragged.

   In the example below, a TextBox control is the control being dragged to (where the drop will occur). The code sets the Text property of the TextBox control equal to the data being dragged.

```
private void textBox1_DragDrop(object sender, Alternet.UI.DragEventArgs e)
{
    textBox1.Text = e.Data.GetData(DataFormats.Text).ToString();
}
```

Clipboard Support

You can implement user cut/copy/paste support and user data transfer to the Clipboard within your AlterNET UI applications by using simple method calls.

The Clipboard class provides methods that you can use to interact with the operating system Clipboard feature. Many applications use the Clipboard as a temporary repository for data. For example, word processors use the Clipboard during cut-and-paste operations. The Clipboard is also useful for transferring data from one application to another.

Add Data to the Clipboard

When you add data to the Clipboard, you can indicate the data format so that other applications can recognize the data if they can use that format. You can also add data to the Clipboard in multiple different formats to increase the number of other applications that can potentially use the data.

A Clipboard format is a string that identifies the format so that an application that uses that format can retrieve the associated data. The DataFormats class provides predefined format names for your use. You can also use your own format names or use the type of an object as its format.

To add data to the Clipboard in one or multiple formats, use the SetDataObject method. You can pass any object to this method, but to add data in multiple formats, you must first add the data to a separate object designed to work with multiple formats. Typically, you will add your data to a DataObject, but you can use any type that implements the IDataObject interface.

> **NOTE**
>
> All applications in an OS environment share the Clipboard. Therefore, the contents are subject to change when you switch to another application.

The following sample illustrates how to add textual data to the Clipboard:

```
private void CopyButton_Click(object sender, System.EventArgs e)
{
    Clipboard.SetText("my string");
}
```

The sample below illustrates how to add data in multiple formats to the Clipboard:

```
private void CopyButton_Click(object sender, System.EventArgs e)
{
```

```
    var data = new DataObject();
    data.SetData(DataFormats.Text, "my text string");
    data.SetData(DataFormats.Files, new[] { "c:\\myfile.txt" });

    Clipboard.SetDataObject(data);
}
```

Retrieve Data from the Clipboard

Some applications store data on the Clipboard in multiple formats to increase the number of other applications that can potentially use the data. A Clipboard format is a string that identifies the format. An application that uses the identified format can retrieve the associated data on the Clipboard. The Data Formats class provides predefined format names for your use. You can also use your own format names or use an object's type as its format.

To determine whether the Clipboard contains data in a particular format, use one of the `Contains`*Format* methods or the GetData method. To retrieve data from the Clipboard, use one of the `Get`*Format* methods or the GetData method.

The following sample illustrates how to retrieve data from the Clipboard:

```
private void PasteButton_Click(object sender, System.EventArgs e)
{
    if (Clipboard.ContainsText)
        MessageBox.Show("Text from the clipboard: " + Clipboard.GetText());

    if (Clipboard.ContainsData(DataFormats.Files))
    {
        string[]? fileNames = Clipboard.GetFiles();
        // Process the file names here.
    }
}
```

Rendering Graphics

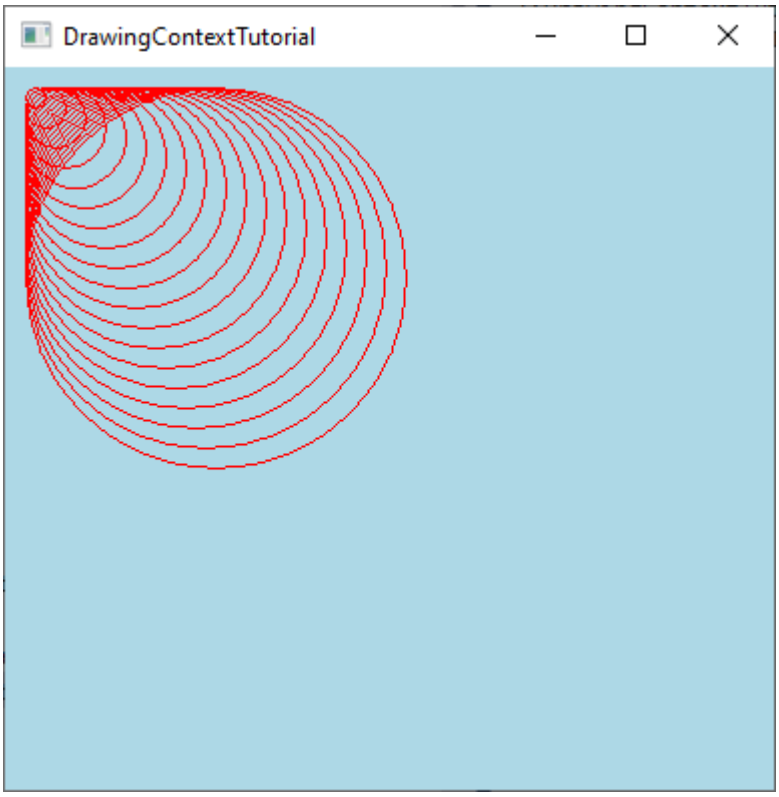
Overview

AlterNET UI includes a set of resolution-independent graphics features that use native rendering on every supported platform.

It supports rendering graphic primitives such as text, images, and graphic shapes with different fonts, pens, and brushes.

The following code example illustrates how graphics can be drawn in a UI element:

```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            e.DrawingContext.FillRectangle(Brushes.LightBlue, e.Bounds);

            for (int size = 10; size < 200; size += 10)
                e.DrawingContext.DrawEllipse(Pens.Red, new(10, 10, size, size));
        }
    }
}
```



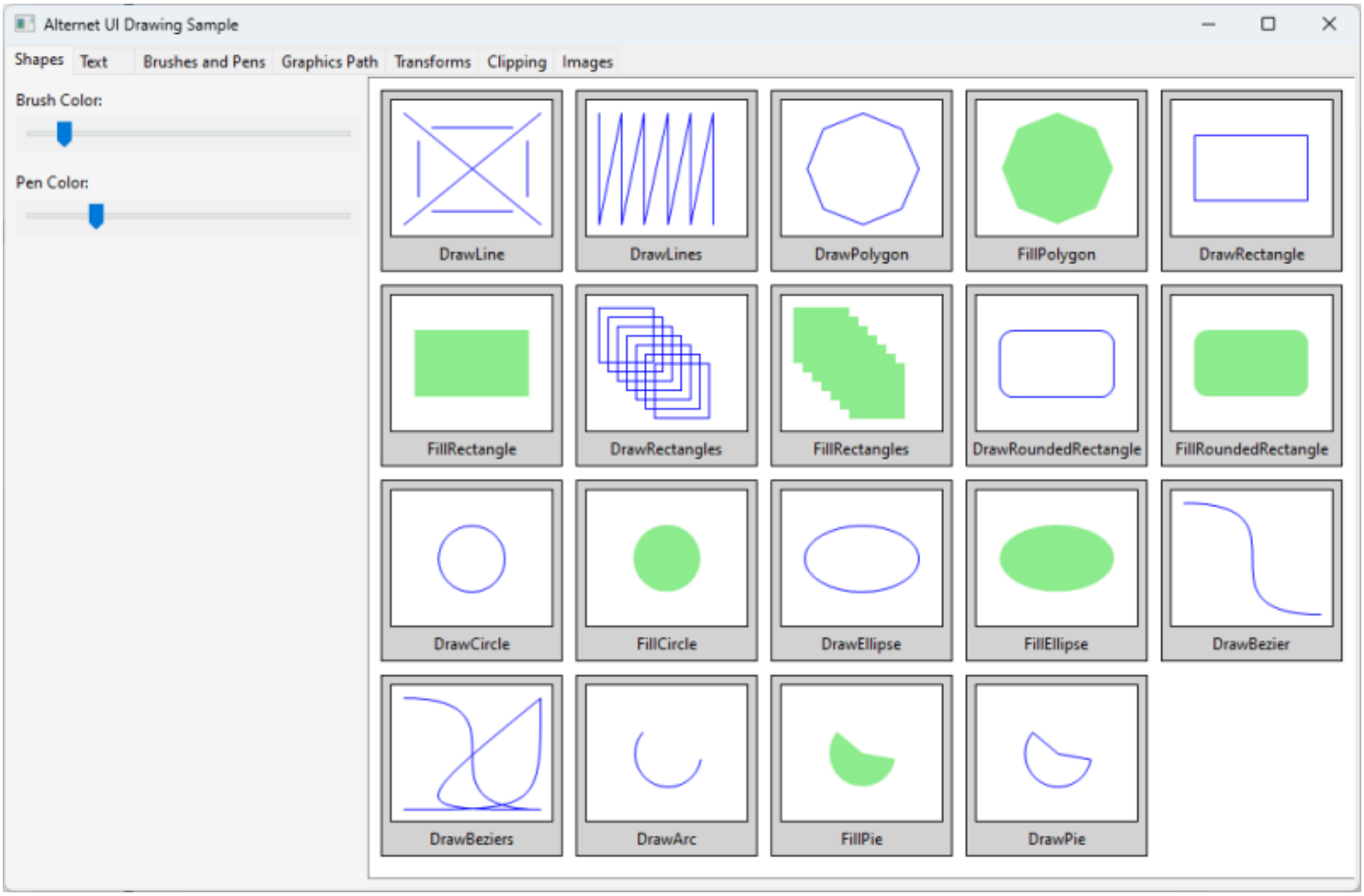
Refer to our blog post to see it in action.

Drawing Context Features

Our Drawing Sample illustrates the features AlterNET UI provides for rendering graphics. Below is a list of the features that Graphics provides, grouped by category. The screenshots are taken from the Drawing Sample.

Geometric Shapes

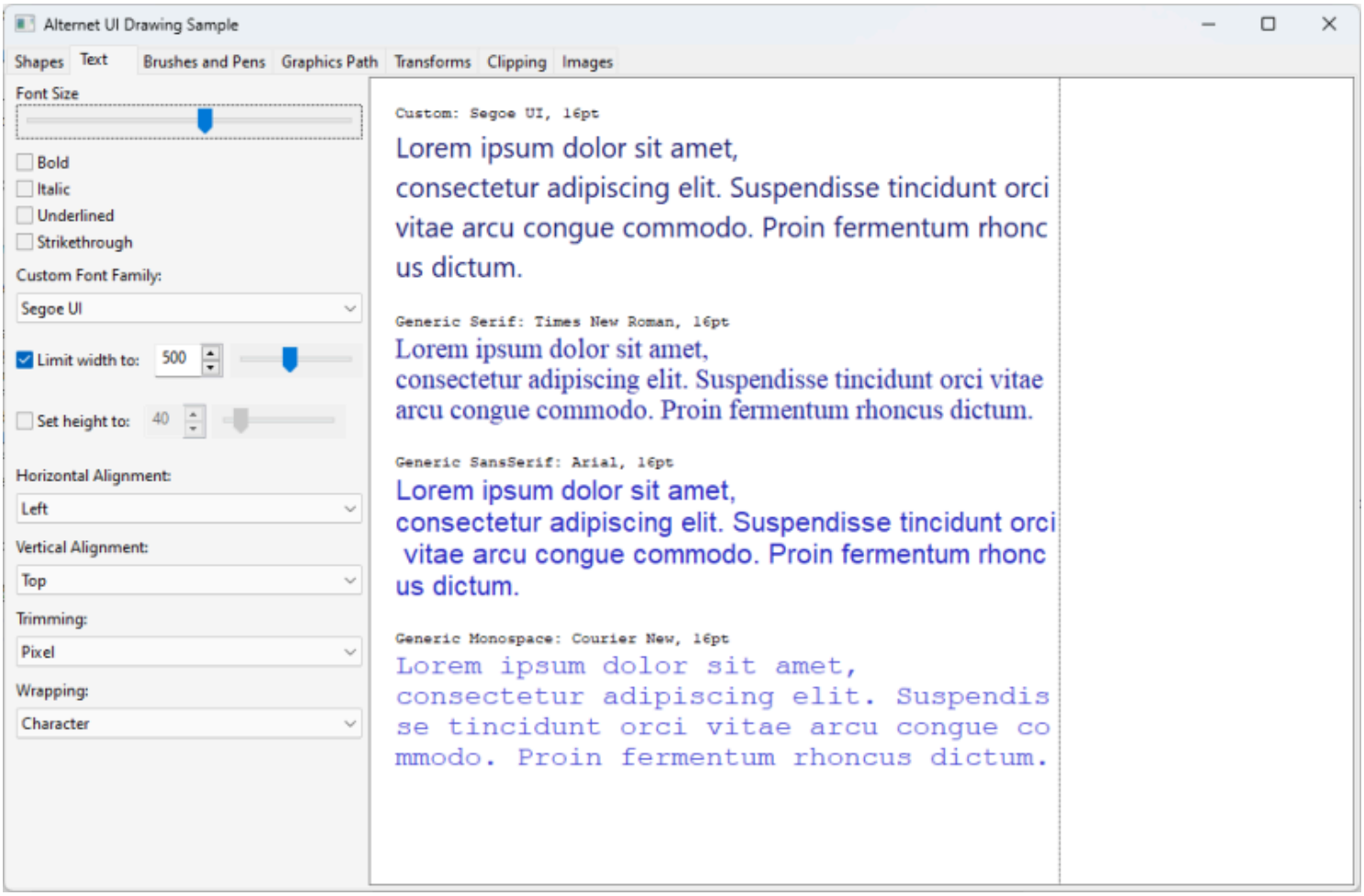
Graphics class provides means to draw a variety of geometric shapes:



- Lines: DrawLine, DrawLines
- Polygons: DrawPolygon, FillPolygon
- Rectangles: DrawRectangle, FillRectangle, DrawRectangles, FillRectangles
- Rounded rectangles: DrawRoundedRectangle, FillRoundedRectangle
- Circles and ellipses: DrawCircle, FillCircle, DrawEllipse, FillEllipse
- Curves: DrawBezier, DrawBeziers
- Arcs and pies: DrawArc, DrawPie, FillPie

Text

Graphics allows to draw text with the specified Font, bounds, and TextWrapping, TextTrimming, TextHorizontalAlignment and TextVerticalAlignment:

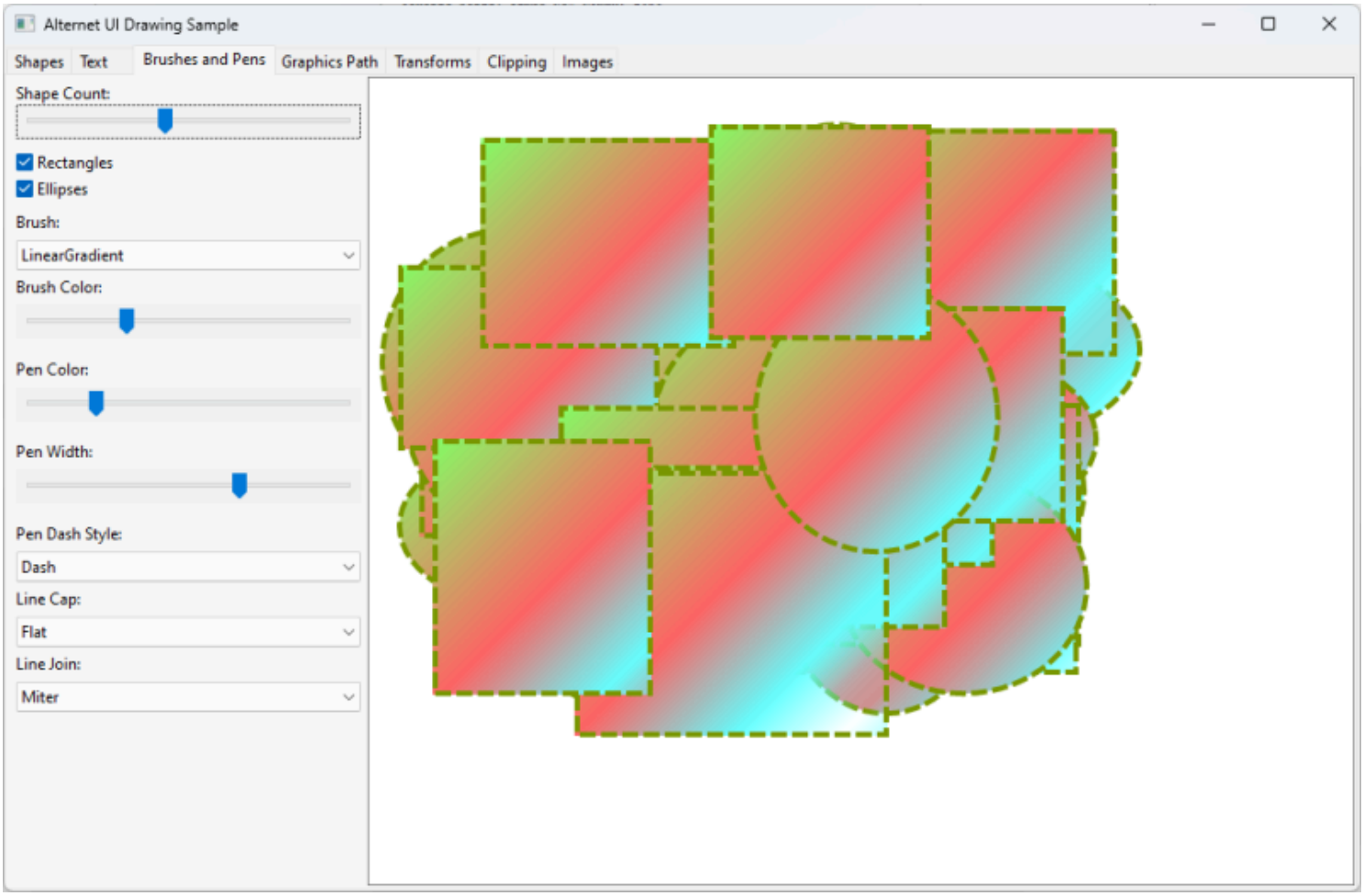


Here is an example of how to draw a wrapped, trimmed, and aligned text string:

```
dc.DrawText(
    "My example text",
    Control.DefaultFont,
    Brushes.Black,
    new Rect(10, 10, 100, 100),
    new TextFormat
    {
        HorizontalAlignment = TextHorizontalAlignment.Center,
        VerticalAlignment = TextVerticalAlignment.Top,
        Wrapping = TextWrapping.Word,
        Trimming = TextTrimming.Character
    });
```

Brushes and Pens

You can draw geometry with different stroke and fill styles provided by the Brush and Pen objects:



Below are the parts of the API responsible for different pen stroke styles:

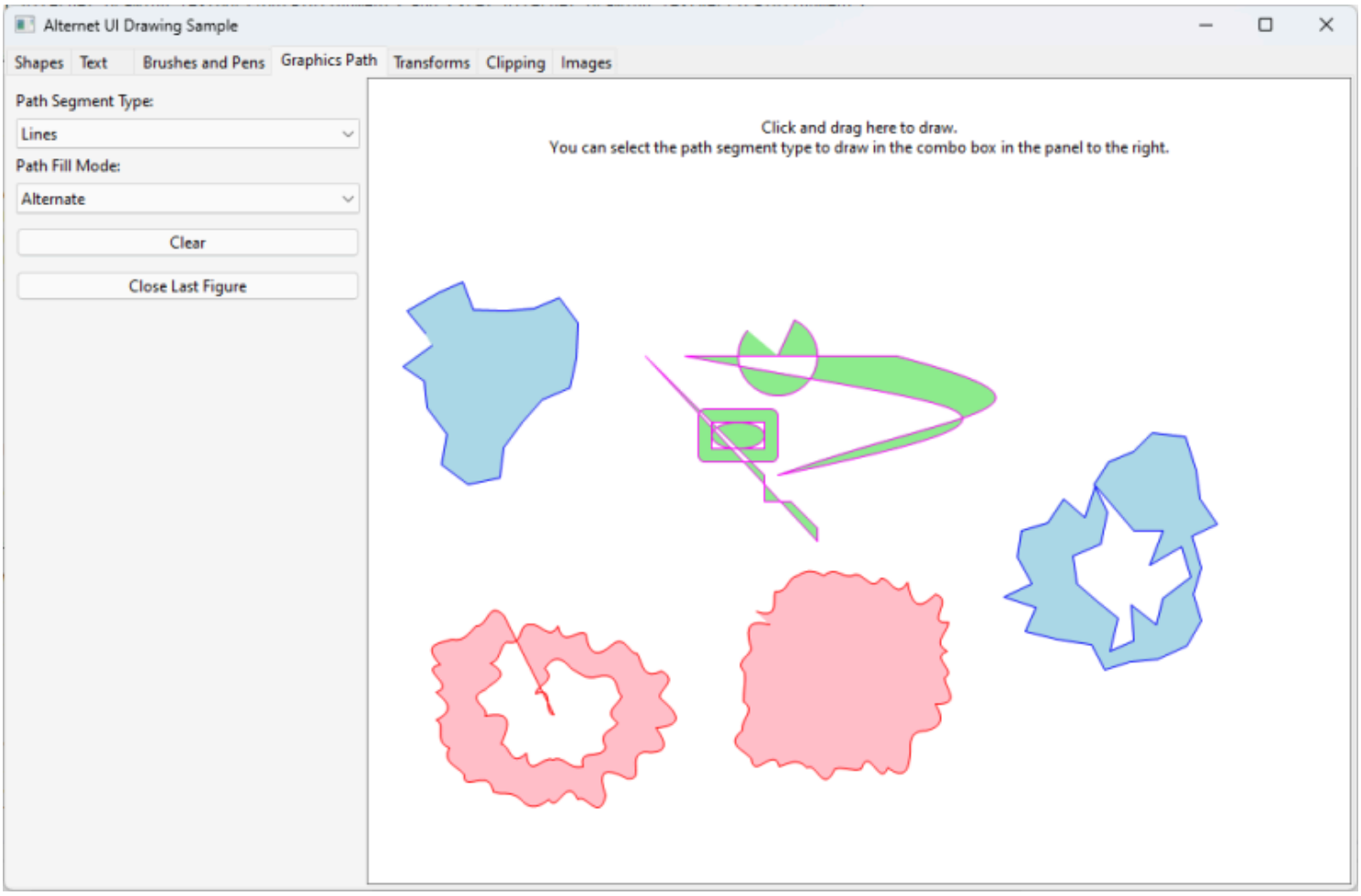
- Solid lines: create an object of the Pen class with a constructor that takes a Color and line thickness value.
- Dashed lines: create an object of the Pen class with a constructor that takes a DashStyle, or set the DashStyle property.
- LineCap and LineJoin enumerations provide different line cap and line join styles.

The following classes allow you to fill geometry with different fill styles:

- Solid fill: use SolidBrush
- Gradient fill: use RadialGradientBrush and LinearGradientBrush
- Pattern fill: use HatchBrush

GraphicsPath

GraphicsPath class provides a way to stroke and fill geometric shapes defined with a series of connected lines and curves:

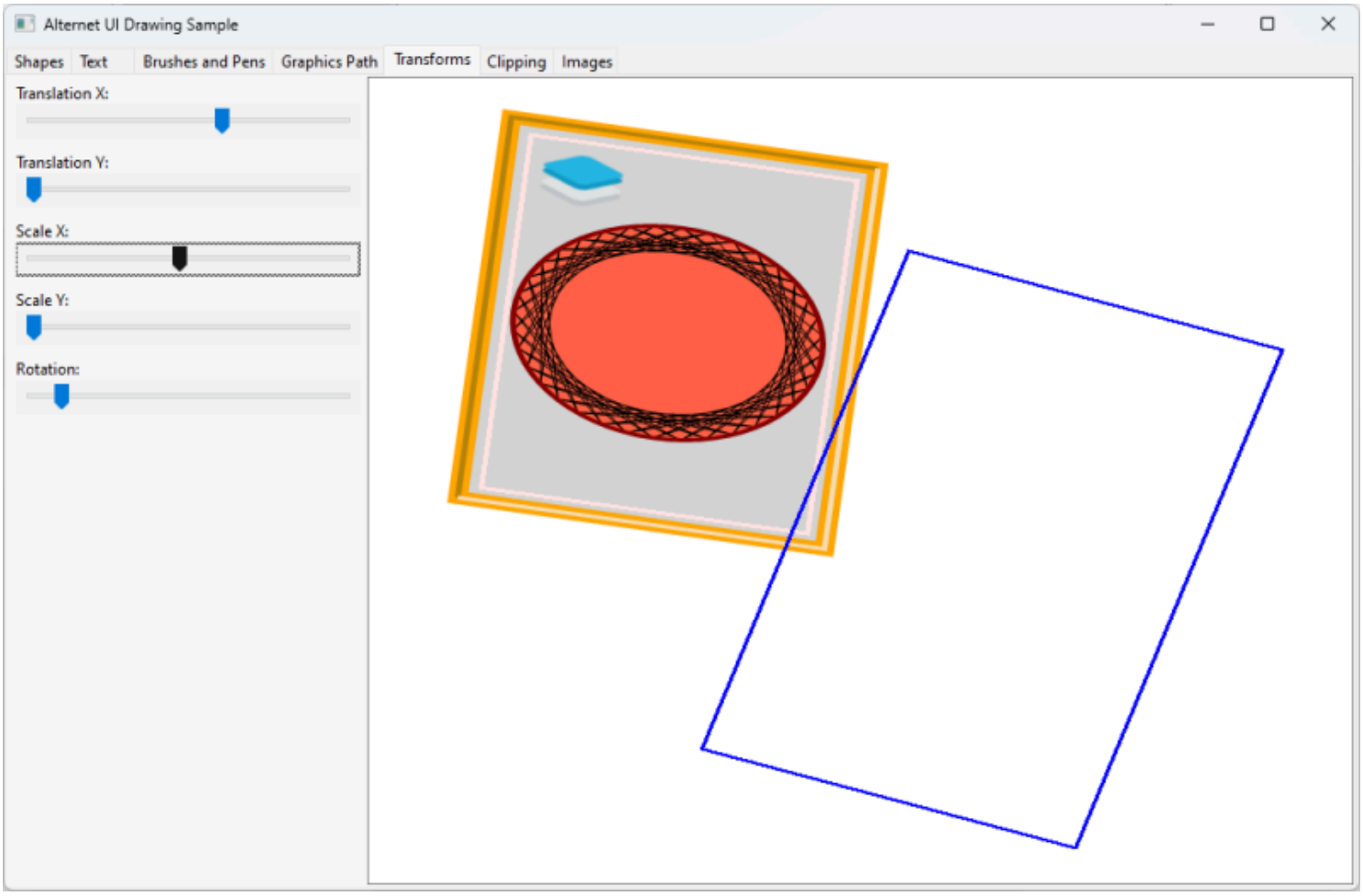


Here are the types of segments supported by the GraphicsPath:

- Lines: AddLine, AddLines, AddLineTo
- Curves: AddBezier, AddBezierTo, AddArc
- Geometric shapes: AddEllipse, AddRectangle, AddRoundedRectangle

Transforms

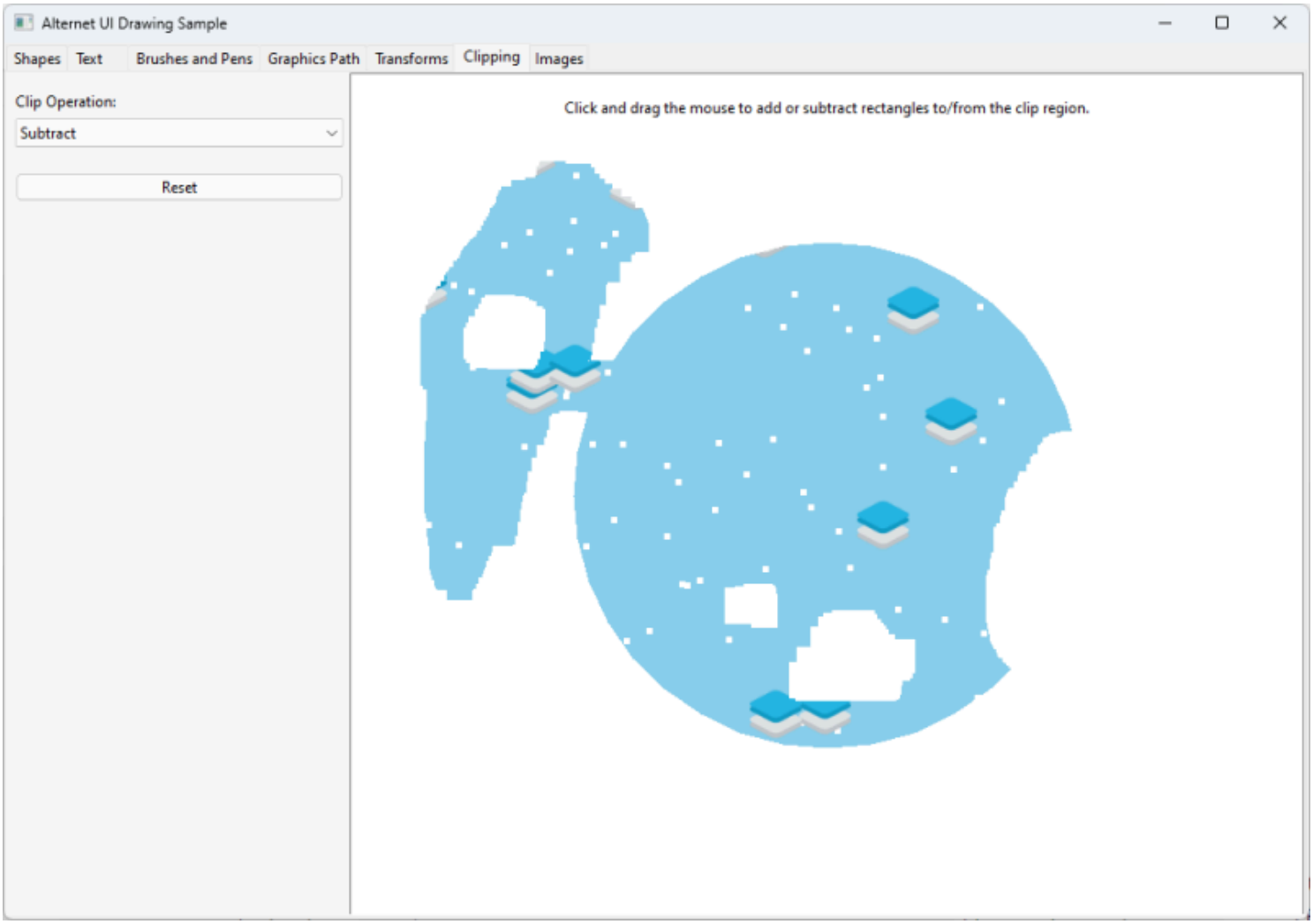
TransformMatrix provides a way to set geometric transform to a Graphics:



The transforms can include translation, rotation, and scale (see the CreateTranslation, CreateRotation and CreateScale methods). Use the Transform property of Graphics to set the current transform. The transforms can be applied sequentially with a stack-like approach, using the PushTransform and Pop methods.

Clip Regions

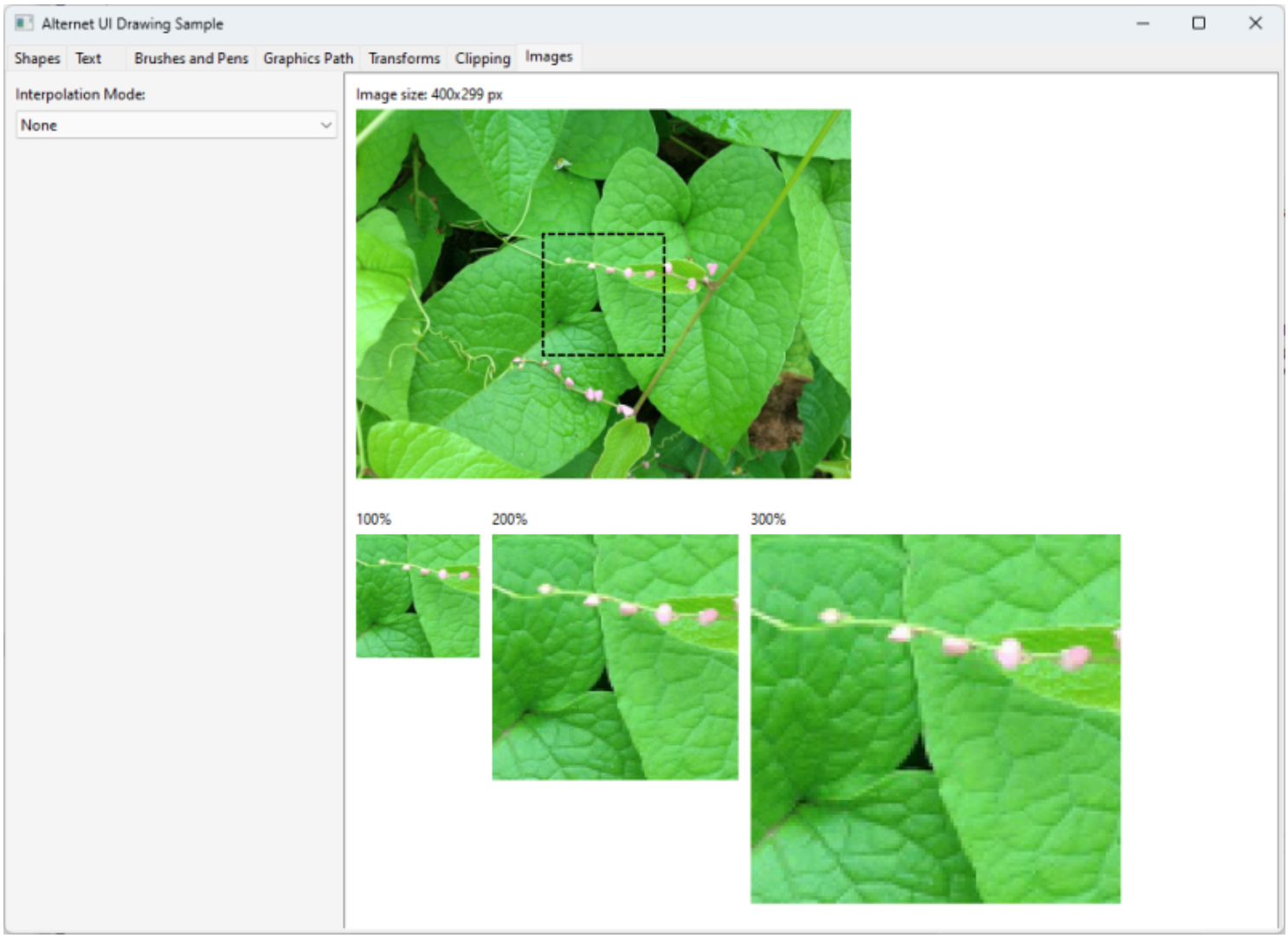
Region class provides a way to set clip region to a Graphics:



Use the Clip property of Graphics to set the current clip region.

Drawing Images

Image class encapsulate a graphical image. DrawImage method overloads provide several ways of drawing images with a specified InterpolationMode:


Alternet UI Drawing Sample
Shapes
Text
Brushes and Pens
Graphics Path
Transforms
Clipping
Images
Interpolation Mode:
None
Image size: 400x299 px
100%
200%
300%

Printing Overview

Printing in AlterNET UI consists primarily of using the PrintDocument component to enable the user to print. The PrintPreviewDialog control, PrintDialog and PageSetupDialog components provide a familiar graphical interface to users.

The `PrintDialog` component is a pre-configured dialog box used to select a printer, choose the pages to print, and determine other print-related settings in UI applications. It's a simple solution for printer and print-related settings instead of configuring your own dialog box. You can enable users to print many parts of their documents: print all, print a selected page range, or print a selection. By relying on standard dialog boxes, you create applications whose basic functionality is immediately familiar to users. The PrintDialog component inherits from the CommonDialog class.

Typically, you create a new instance of the PrintDocument component and set the properties that describe what to print using the PrinterSettings and PageSettings classes. Calling the Print method prints the document.

Working with the component

Use the PrintDialog.ShowModal method to display the dialog at run time. This component has properties that relate to either a single print job (PrintDocument class) or the settings of an individual printer (PrinterSettings class). One of the two, in turn, may be shared by multiple printers.

How to capture user input from a PrintDialog at run time

You can set options related to printing at design time. Sometimes you may want to change these options at run time, most likely because of choices made by the user. You can capture user input for printing a document using the PrintDialog and the PrintDocument components. The following steps demonstrate displaying the print dialog for a document:

1. Add a PrintDialog and a PrintDocument component to your form.
2. Set the Document property of the PrintDialog to the PrintDocument added to the form.

```
printDialog1.Document = printDocument1;
```

3. Display the PrintDialog component by using the ShowModal method.

```
// display show dialog, and if the user selects "Ok" the document is printed
if (printDialog1.ShowDialog() == DialogResult.OK)
    printDocument1.Print();
```

4. The user's printing choices from the dialog will be copied to the PrinterSettings property of the PrintDocument component.

How to create print jobs

The foundation of printing in AlterNET UI is the PrintDocument component, more specifically, the PrintPage event. By writing code to handle the PrintPage event, you can specify what to print and how to print it. The following steps demonstrate creating a print job:

1. Add a PrintDocument component to your form.

2. Write code to handle the PrintPage event.

   You'll have to code your own printing logic. Additionally, you'll have to specify the material to be printed.

   As a material to print, in the following code example, a sample graphic in the shape of a red rectangle is created in the PrintPage event handler.

```
private void PrintDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e) =>
    e.Graphics.FillRectangle(Brushes.Red, new Rectangle(100, 100, 100, 100));
```

You can also write code for the BeginPrint and EndPrint events. It will help to include an integer representing the total number of pages to print that is decremented as each page prints.

For more information about the specifics of AlterNET UI print jobs, including how to create a print job programmatically, see PrintPageEventArgs.

Using Resource URIs

In AlterNET UI, uniform resource identifiers (URIs) are used to identify and load files in the following scenarios:

- Loading images.
- Loading data files.
- Any other scenario when read-only access to a resource file is required.

Using `embres:` Scheme

`embres:` scheme is used to load an embedded resource from an assembly. The URIs in this scheme have the following format:

```
embres:Manifest.Resource.Name[?assembly=assembly-name]
```

The following is an example of using an image from a resource embedded into the current assembly :

```
<PictureBox Image="embres:EmployeeFormSample.Resources.EmployeePhoto.jpg" />
```

The resource in the example above is embedded into the assembly in the following way (an excerpt from the `.csproj` file):

```
<ItemGroup>
    <EmbeddedResource Include="Resources\EmployeePhoto.jpg" />
</ItemGroup>
```

The `EmployeeFormSample` part of the manifest resource name comes from the assembly *root namespace*, which is the same as the assembly name by default.

Using `file:` Scheme

`file:` scheme is used to load a file. The URIs in this scheme have the following format:

```
file://<host>/<path>
```

Linux:

These urls point to the same file **/etc/fstab**:

```
file://localhost/etc/fstab
file:///etc/fstab
file:///etc/./fstab
file:///etc/../etc/fstab
```

Mac OS:

These urls point to the same file **/var/log/system.log**:

```
file://localhost/var/log/system.log
file:///var/log/system.log
```

Windows:

These urls point to the same file **c:\WINDOWS\clock.avi**:

```
file://localhost/c|/WINDOWS/clock.avi
file:///c|/WINDOWS/clock.avi
file://localhost/c:/WINDOWS/clock.avi
file:///c:/WINDOWS/clock.avi
```

Focus Management

AlterNET UI includes several API elements to control input focus.

Keyboard focus refers to the object that is receiving keyboard input. The element with keyboard focus has Focused set to true. There can be only one element with keyboard focus on the entire desktop.

Keyboard focus can be obtained through user interaction with the UI, such as tabbing to an element or clicking the mouse on certain elements. Keyboard focus can also be obtained programmatically by using the SetFocus method.

The SetFocus method returns true if the control successfully received input focus. The control can have the input focus while not displaying any visual cues of having the focus. This behavior is primarily observed by the nonselectable controls listed below, or any controls derived from them.

When the user presses the TAB key, the input focus is set to the next control in the tab order. Controls with the TabStop property value of false are not included in the collection of controls in the tab order.

When you change the focus by using the keyboard (TAB, SHIFT+TAB, and so on), by calling the SetFocus or FocusNextControl methods focus events occur in the following order:

- GotFocus
- LostFocus

System Requirements

AlterNET UI supports the following .NET versions:

- .NET 6.0 or newer
- .NET Framework 4.62 or newer

The following operating systems are supported:

- Windows 7 Service Pack 1 or newer
- macOS 10.15 or newer
- Linux (different distributions)

See this page for detailed information.

Microsoft Visual Studio 2022 is supported by the **AlterNET UI Visual Studio Extension**.

AlterNET UI also supports a development workflow with **Visual Studio Code** and `dotnet` command line tools.

"Hello, World" with Visual Studio

In this tutorial, you will create a cross-platform desktop application using C# and Microsoft Visual Studio. The application will display a message box in response to a button click.

Microsoft Visual Studio is supported on Windows OS only. To develop on macOS or Linux, see "Hello, World" with Command-Line and VS Code.

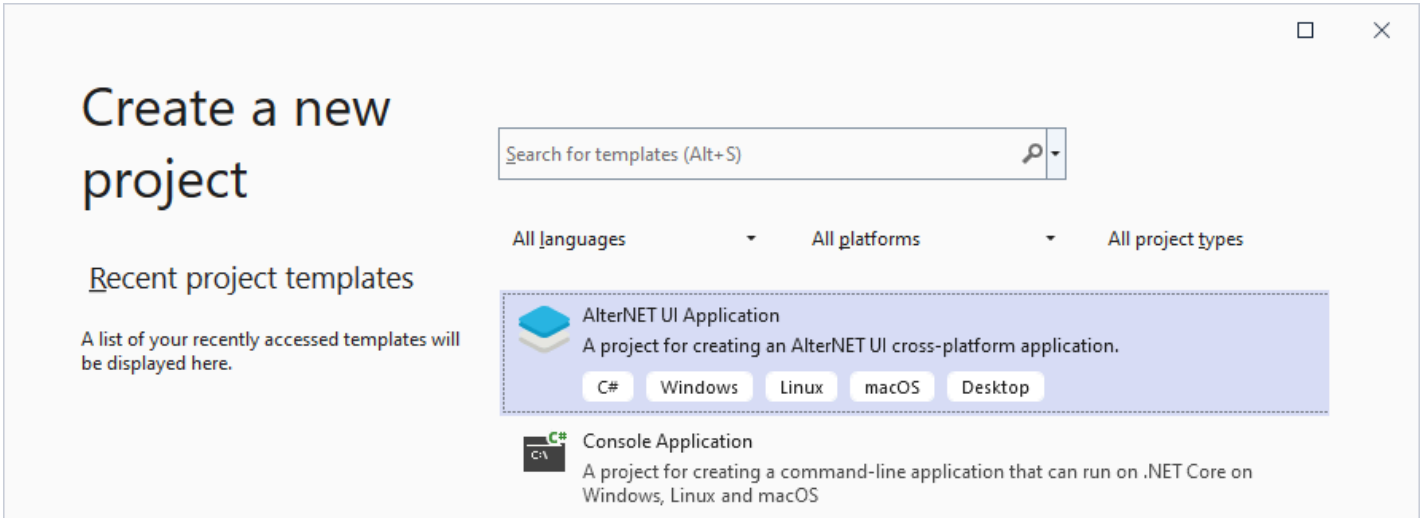
Prerequisites

1. Install Microsoft Visual Studio. Visual Studio 2022 is supported.
2. Ensure the ".NET Desktop Development" or "ASP.NET and web development" workflow is installed.
3. Download and install AlterNET UI Visual Studio extension.

Create New Project

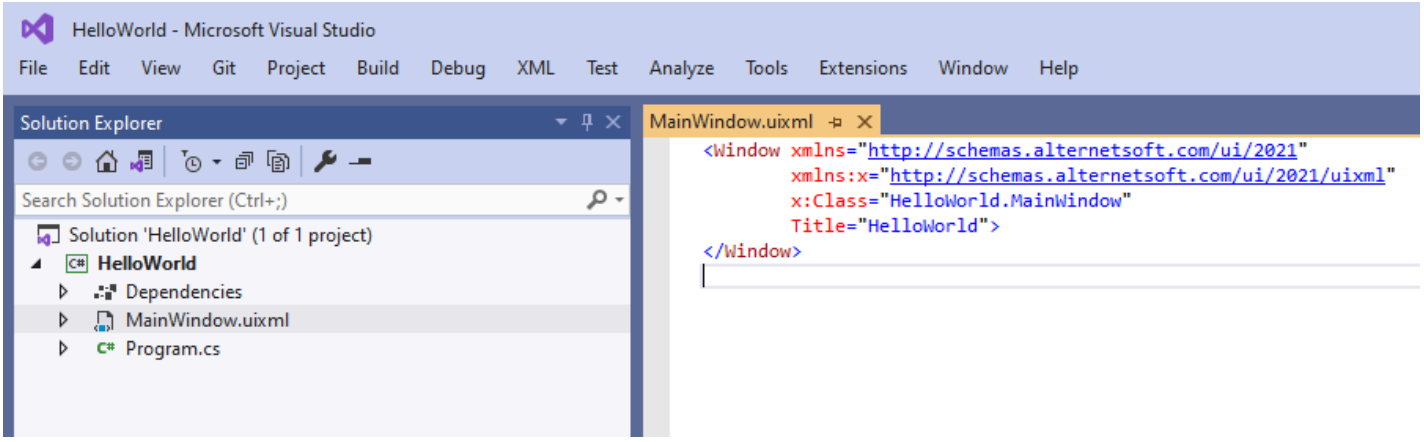
1. Open Visual Studio, and in the start window, select **Create new project**.

2. On the **Create new project** page, locate the AlterNET UI Application template. Select it, then click **Next**.

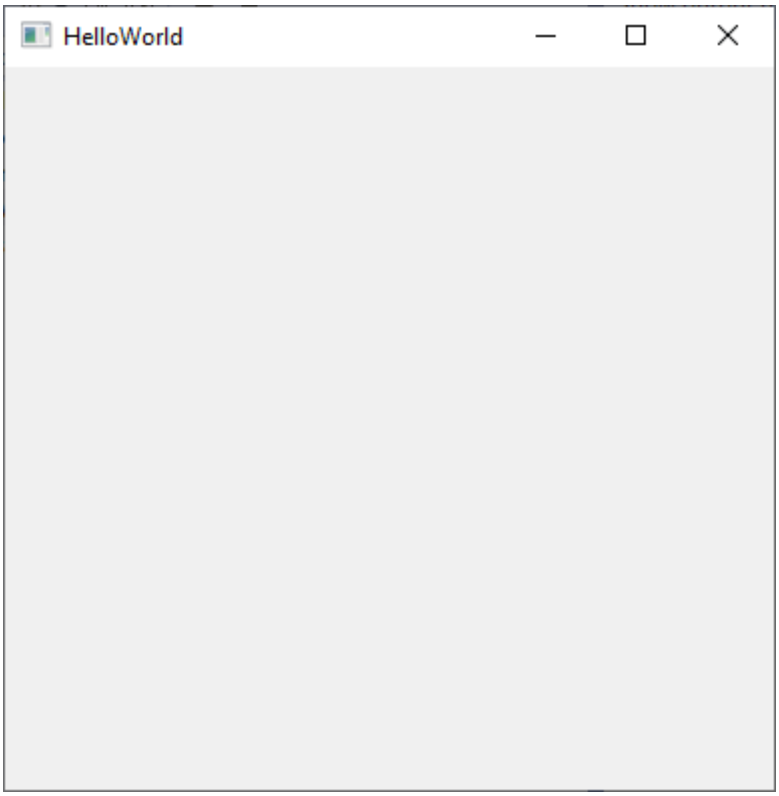


3. On the **Configure your new project** page set the project name to `HelloWorld` and specify the desired project location. When done, click **Create**.

4. The project will be created, and you will be presented with a development environment.



5. Press `Ctrl+F5` to build and run the application. The application will start and display its window:



6. In Visual Studio, open `MainWindow.uixml`. In the editor, change the `Title` attribute value from `"HelloWorld"` to `"My First Application"`:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
</Window>
```

7. Press `Ctrl+F5` to build and run the application and see its window title has changed accordingly.

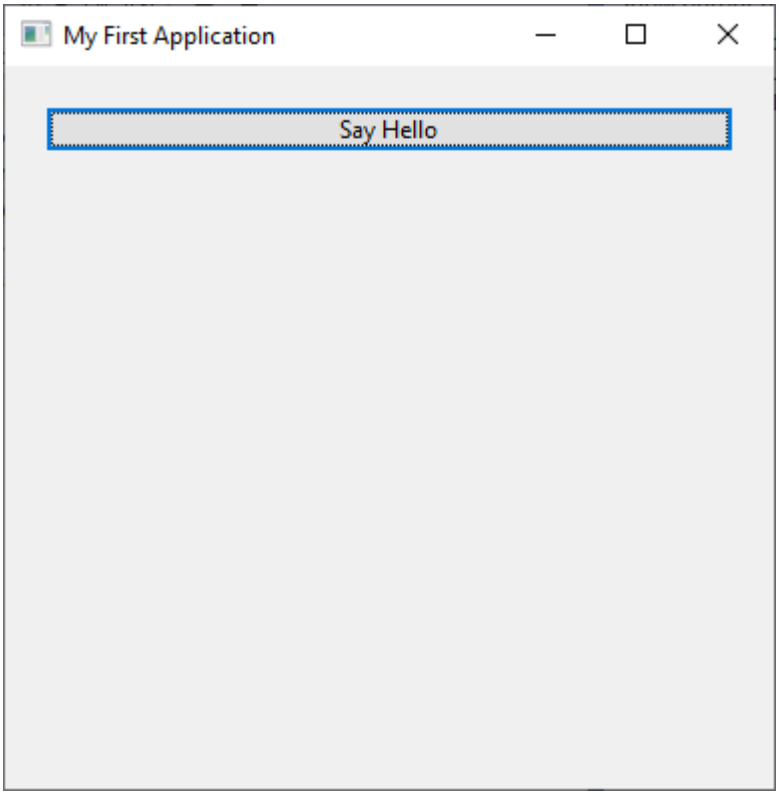
> **NOTE**
>
> By default, the created project will use .NET 6.0 as a target framework. If .NET 6.0 runtime is not installed on your machine, you will be prompted to do so on the first application run.

Add Button to the Window

1. In `MainWindow.uixml`, add the following markup:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20" />
  </StackPanel>
</Window>
```

2. Run the application by pressing `Ctrl+F5`:



Write Code to Respond to the Button Click

1. In `MainWindow.uixml`, add the `Click` attribute to the `Button` element like the following:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20" Click="HelloButton_Click" />
  </StackPanel>
</Window>
```

This will bind the `Click` event to its handler, ' HelloButton_Click`.

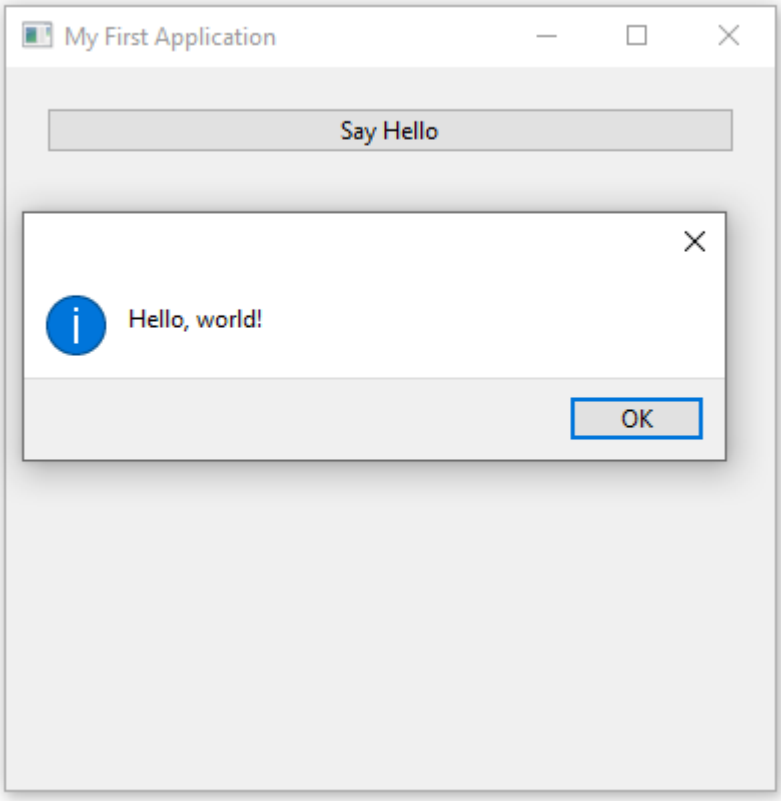
2. In `MainWindow.uixml.cs` file, add the following `HelloButton_Click` method:

```
using System;
using Alternet.UI;

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

        private void HelloButton_Click(object? sender, EventArgs e)
        {
            MessageBox.Show("Hello, world!");
        }
    }
}
```

3. You can use IntelliSense features provided by the AlterNET UI Visual Studio Extension:

```
MainWindow.uixml.cs  MainWindow.uixml
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20"/>
  </StackPanel>
</Window>
```

4. Run the application, then click **Say Hello** button. The message box appears:



> **NOTE**
>
> The application created in this tutorial can be compiled and run without modifications on all the supported platforms: Windows, macOS, and Linux.

Congratulations, you have successfully completed the "Hello, World" tutorial using Microsoft Visual Studio.

For a similar tutorial, but using command line tools and Visual Studio Code, see "Hello, World" with Command-Line and Visual Studio Code.

"Hello, World" with Command-Line and Visual Studio Code

In this tutorial, you will create a cross-platform desktop application using C#, .NET command line tools, and Visual Studio Code. The application will display a message box in response to a button click.

Prerequisites

1. Download and install .NET SDK. The minimum supported SDK version is .NET 6.0.
2. Install AlterNET UI project templates by running

   ```
   dotnet new install Alternet.UI.Templates
   ```

3. Download and install Visual Studio Code
4. In Visual Studio Code, ensure the C# extension is installed. For information about how to install extensions on Visual Studio Code, see VS Code Extension Marketplace.
5. If you develop under Linux, please install required packages as described at the end of this page.

Create New Project

1. Create a new directory for your application; name it `HelloWorld`
2. Open the **Command Prompt** window (**Terminal** on macOS or Linux)
3. Navigate the terminal to the created directory:

   ```
   cd path/to/HelloWorld
   ```

4. Enter the following command to create a new project in the current directory:

   ```
   dotnet new alternet-ui
   ```

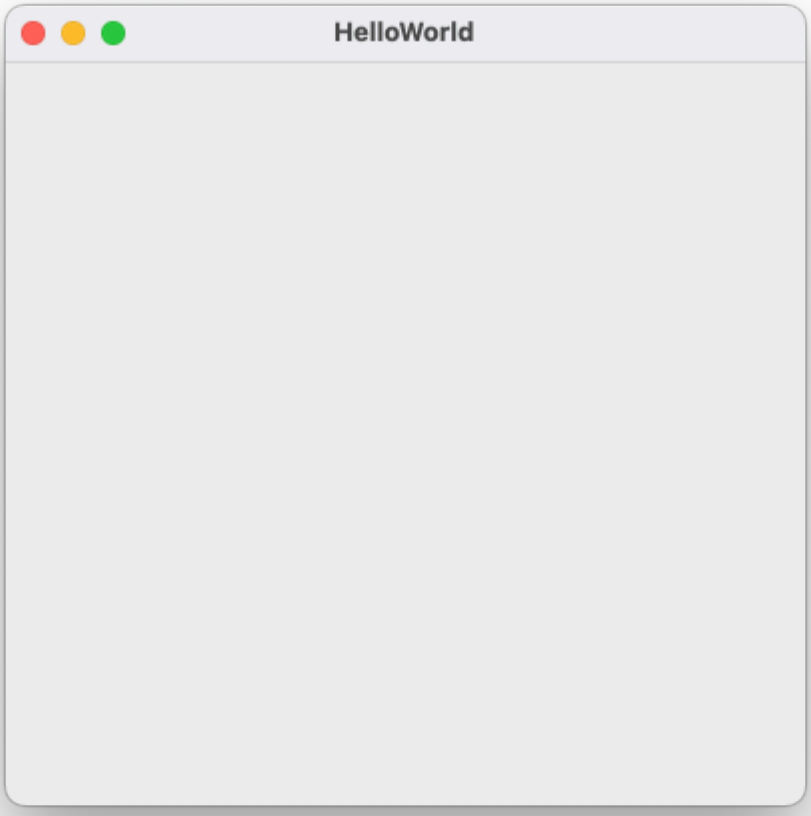
5. The following files will be created:

   ```
   HelloWorld.csproj
   MainWindow.uixml
   MainWindow.uixml.cs
   Program.cs
   ```

6. Compile and run the created project by executing:

   ```
   dotnet run
   ```

The application will start and display its window:



> **NOTE**
>
> By default, the created project will use .NET 6.0 as a target framework. If .NET 6.0 runtime is not installed on your machine, you will be prompted to do so on the first application run.

> **NOTE**
>
> You can also create .cs/.uixml files for a new window from the console like this:
>
> ```
> dotnet new alternet-ui-window -n MyNewWindow --namespace Test1
> ```
>
> Where `MyNewWindow` is a name for a new window class, and `Test1` is the created class namespace name.

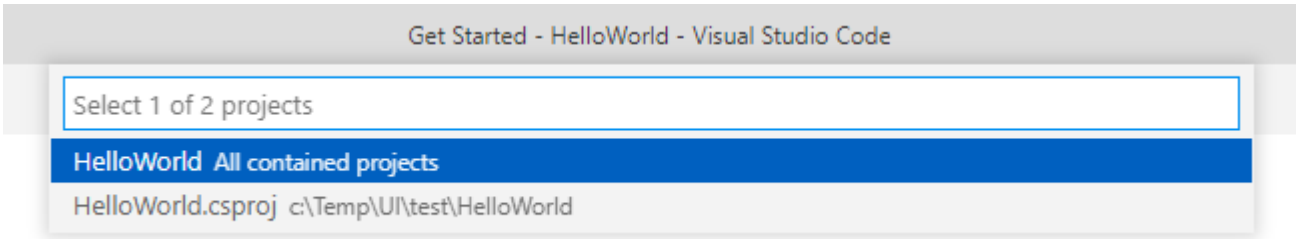
Open Project with Visual Studio Code

1. Start Visual Studio Code.

2. Select **File** > **Open Folder** (**File** > **Open...** on macOS) from the main menu.

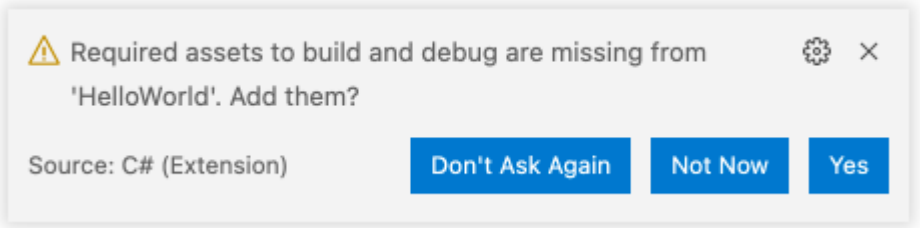
3. In the **Open Folder** dialog, locate a *HelloWorld* folder and click **Select Folder** (**Open** on macOS).

4. The popup prompting **Select 1 of 2 projects** will appear at the top of the screen:



   Select **All contained projects**.

5. After several seconds, a popup dialog with the message **Required assets to build and debug are missing from 'HelloWorld'. Add them?** will appear at the bottom-right corner of the screen:



   Select **Yes**. The `.vscode` subdirectory will be created with the workspace settings automatically set up.

6. Now, you can debug your application by pressing `F5` or run it without debugging by pressing `Ctrl+F5`. The application will be built automatically if required.

7. Open `MainWindow.uixml` by clicking the corresponding item in the VS Code **Explorer** panel. In the editor, change the `Title` attribute value from `"HelloWorld"` to `"My First Application"`:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
</Window>
```

8. Press `Ctrl+F5` to build and run the application and see its window title has changed accordingly.

> **NOTE**
>
> For information and tutorials on general C# development and debugging with Visual Studio Code, see the corresponding MSDN article.

Uixml Syntax Highlight

In order to have syntax highlight in uixml, you need:

- Create '.vscode' subfolder in you project folder.
- Create there 'settings.json' file.
- Edit 'settings.json' and add there 'files.associations' section:

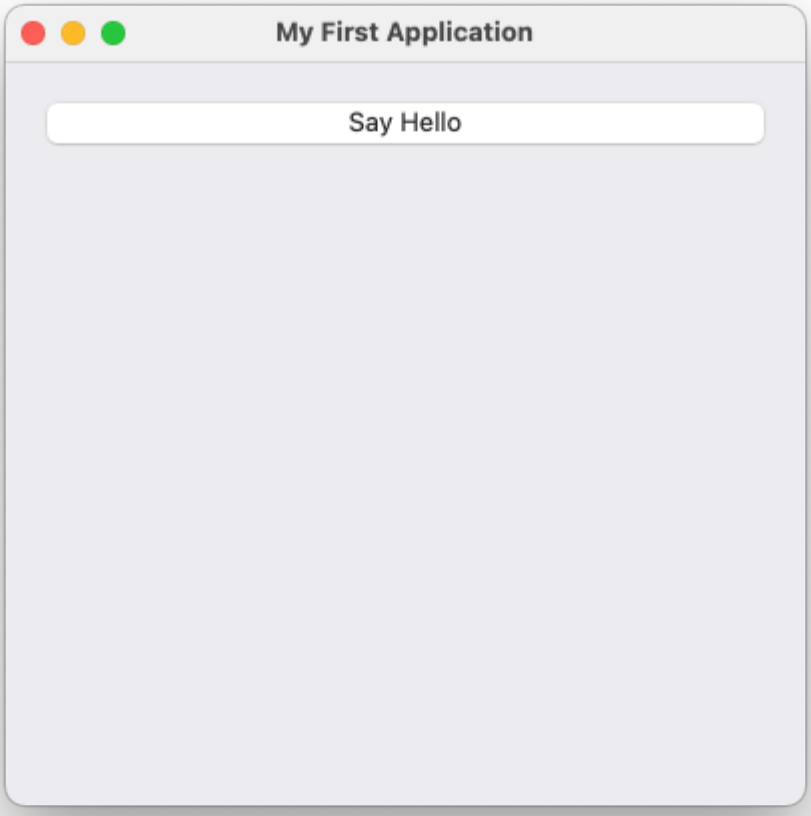
```
{
        "files.associations": {
            "*.uixml": "xml",
        }
}
```

Add Button to the Window

1. In `MainWindow.uixml`, add the following markup:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20" />
  </StackPanel>
</Window>
```

2. Run the application by pressing `Ctrl+F5`:



Write Code to Respond to the Button Click

1. In `MainWindow.uixml`, add the `Click` attribute to the `Button` element like the following:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="HelloWorld.MainWindow"
        Title="My First Application">
  <StackPanel>
    <Button Name="helloButton" Text="Say Hello" Margin="20" Click="HelloButton_Click" />
  </StackPanel>
</Window>
```

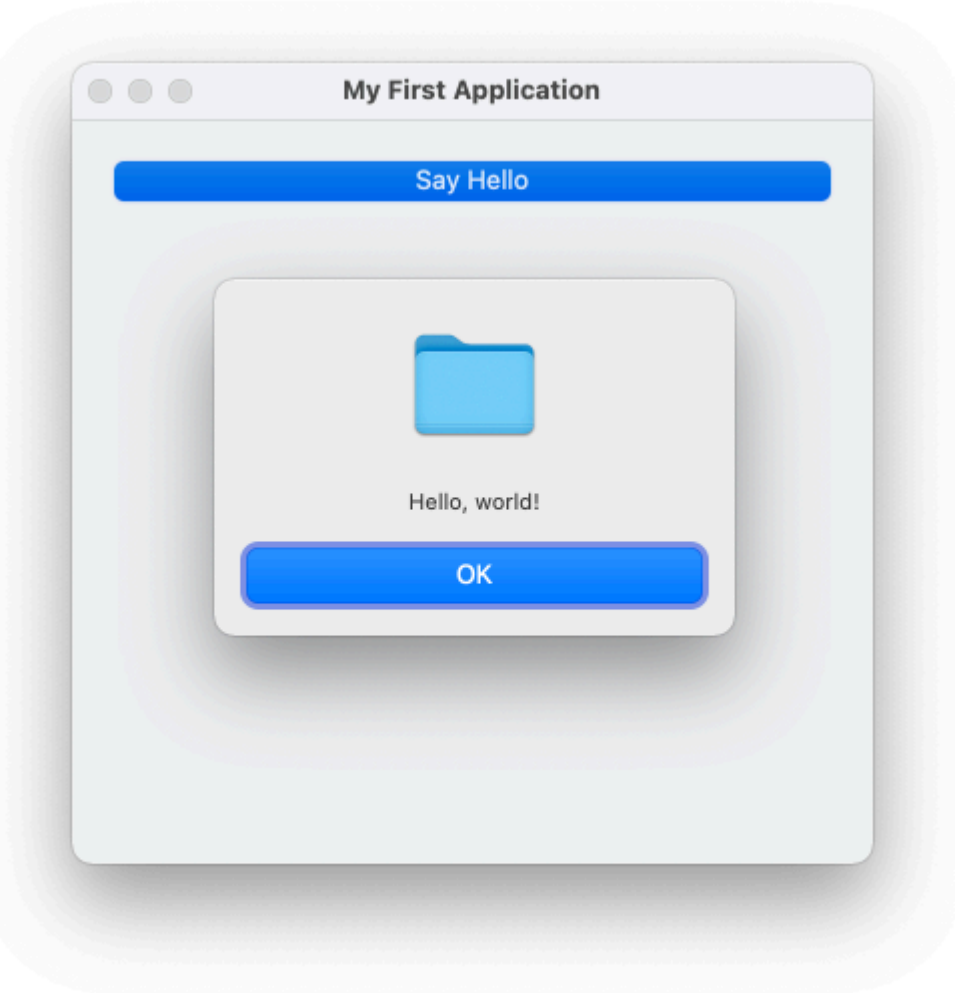
   This will bind the `Click` event to its handler, ' HelloButton_Click`.

2. In `MainWindow.uixml.cs` file, add the following `HelloButton_Click` method:

```
using System;
using Alternet.UI;
```

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

        private void HelloButton_Click(object? sender, EventArgs e)
        {
            MessageBox.Show("Hello, world!");
        }
    }
}
```

3. Run the application, then click **Say Hello** button. The message box appears:



> **NOTE**
>
> The application created in this tutorial can be compiled and run without source code modifications on all the supported platforms: Windows, macOS, and Linux.

Linux

Before running Alternet.UI applications on Linux, you need to install required packages. There is special installation script for Ubuntu. You can download it from the GitHub repository (File: Install.Scripts/Ubuntu.Install.Packages.sh). This is development packages, end-users do not need to install all of them.

Congratulations, you have completed the "Hello, World" tutorial using command line tools and Visual Studio Code.

For a similar tutorial but using Visual Studio on Windows, see "Hello, World" with Visual Studio.

Rendering with Graphics

This tutorial will teach you how to create a custom Control, which draws itself on the screen using Graphics class.

1. Create a new AlterNET UI Application project, name it `DrawingContextTutorial`. For step-by-step guidance on how to create a new AlterNET UI project, see "Hello, World" Tutorial.

2. Add a new empty class named `DrawingControl` to the project. Make the class `public`, and derive it from Control:

```
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
    }
}
```

3. Open `MainWindow.uixml`. Add the reference to the local namespace, and add a `DrawingControl` to the window:

```
<Window xmlns="http://schemas.alternetsoft.com/ui/2021"
        xmlns:x="http://schemas.alternetsoft.com/ui/2021/uixml"
        x:Class="DrawingContextTutorial.MainWindow"
        Title="DrawingContextTutorial"
        xmlns:local="clr-
namespace:DrawingContextTutorial;assembly=DrawingContextTutorial">
  <local:DrawingControl />
</Window>
```

4. Compile and run the application. An empty window will appear. This is because `DrawingControl` does not paint itself yet.

5. In the `DrawingControl` class, add a default constructor. In its body, set UserPaint property to `true`. Also, override the OnPaint method:

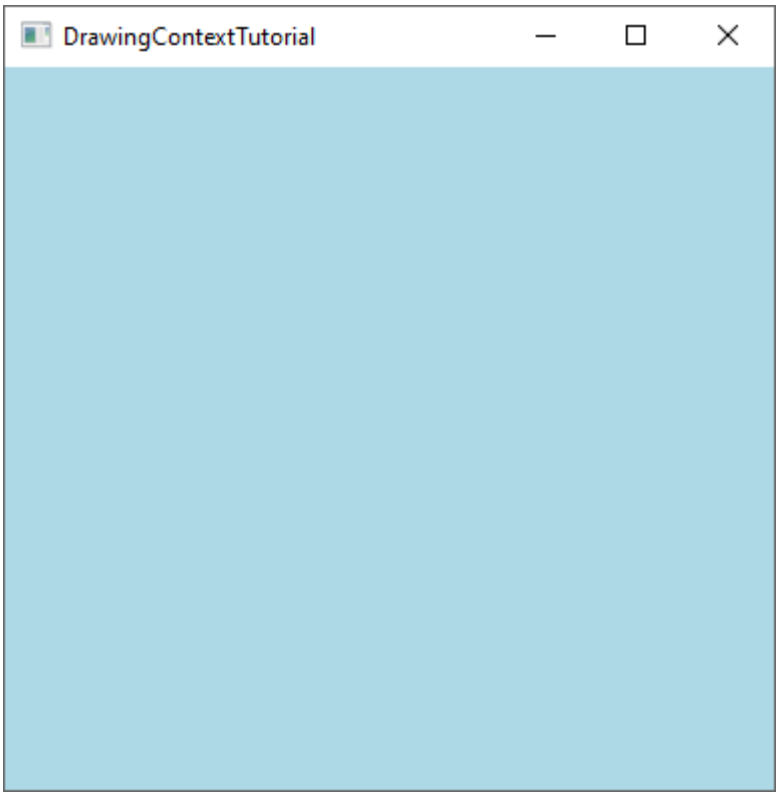
```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
```

```
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
        }
    }
}
```

6. In the overridden `OnPaint` method, add the following Graphics.FillRectangle call to paint the control's background LightBlue:

```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            e.DrawingContext.FillRectangle(Brushes.LightBlue, e.Bounds);
        }
    }
}
```

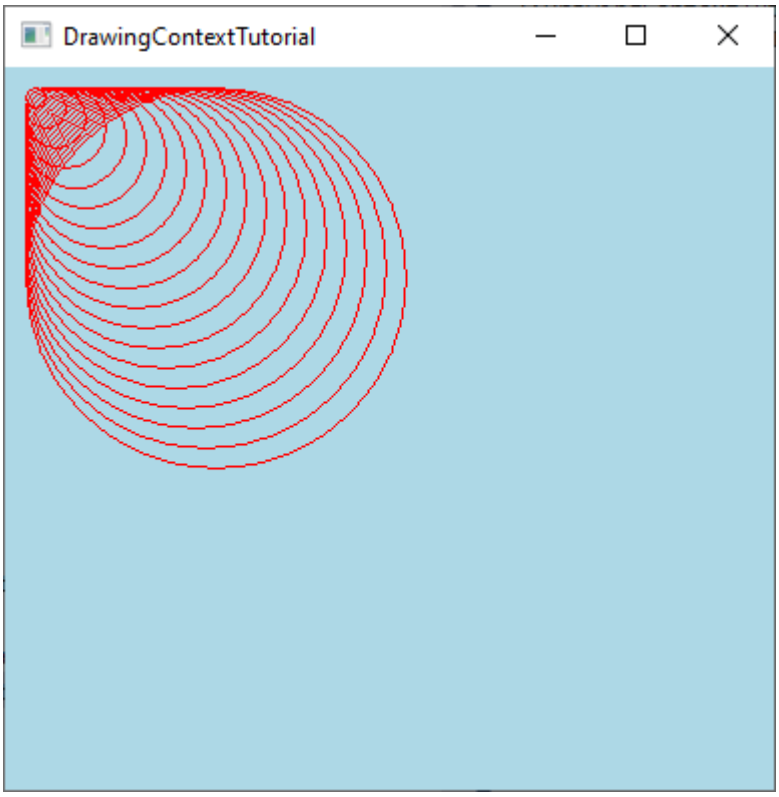
7. Build and run the application. The displayed window will have a light blue background:



8. In the overridden `OnPaint` method, add the following two lines of code to paint a red circular pattern:

```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            e.DrawingContext.FillRectangle(Brushes.LightBlue, e.Bounds);

            for (int size = 10; size < 200; size += 10)
                e.DrawingContext.DrawEllipse(Pens.Red, new(10, 10, size, size));
        }
    }
}
```

9. Build and run the application. The displayed window will look like the following:

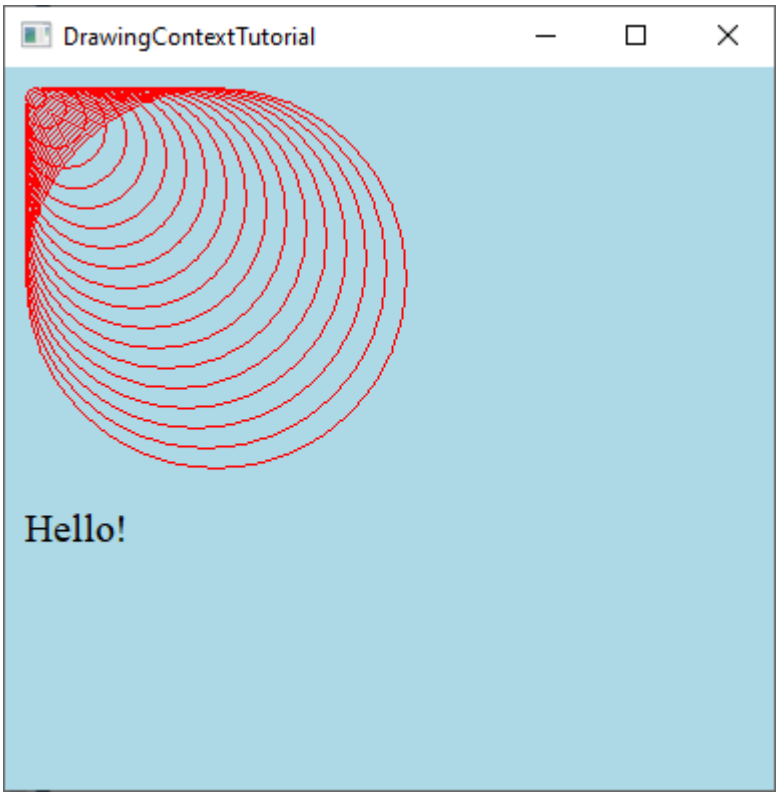


10. Now, let's draw a simple line of text. To do that, we will create a cached Font instance, and draw a text line using the Graphics.DrawText method:

```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        private static Font font = new Font(FontFamily.GenericSerif, 15);

        protected override void OnPaint(PaintEventArgs e)
        {
            e.DrawingContext.FillRectangle(Brushes.LightBlue, e.Bounds);

            for (int size = 10; size < 200; size += 10)
                e.DrawingContext.DrawEllipse(Pens.Red, new(10, 10, size, size));

            e.DrawingContext.DrawText("Hello!", font, Brushes.Black, new
PointF(10, 220));
        }
```

```
    }
}
```

11. Build and run the application. The displayed window will look like the following:

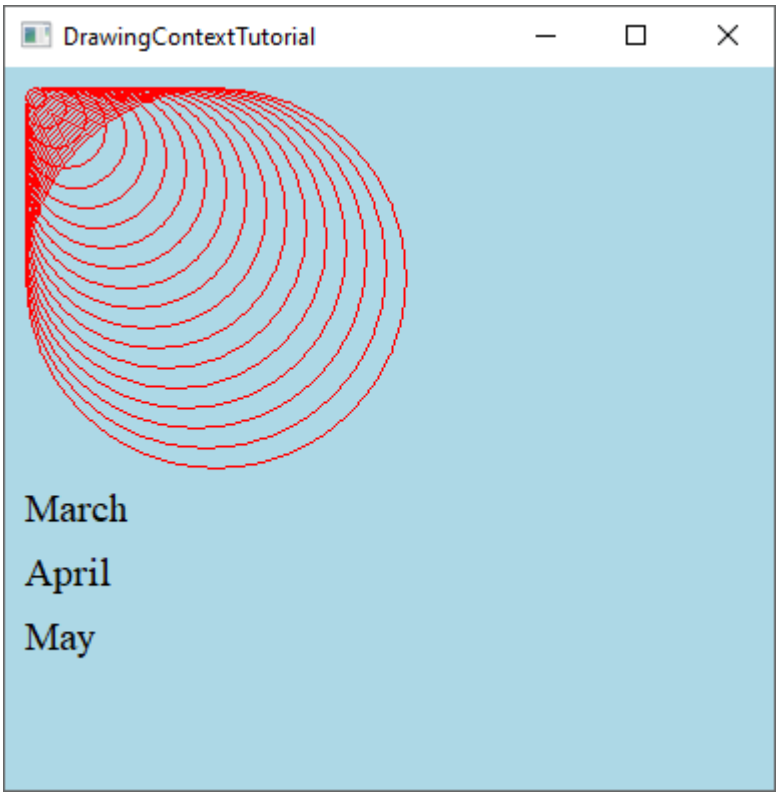


12. To demonstrate how Graphics.MeasureText works, let's draw the names of the three spring months one under another:

```
using Alternet.Drawing;
using Alternet.UI;

namespace DrawingContextTutorial
{
    public class DrawingControl : Control
    {
        public DrawingControl()
        {
            UserPaint = true;
        }

        private static Font font = new Font(FontFamily.GenericSerif, 15);

        protected override void OnPaint(PaintEventArgs e)
        {
            e.DrawingContext.FillRectangle(Brushes.LightBlue, e.Bounds);
```

```
            for (int size = 10; size < 200; size += 10)
                e.DrawingContext.DrawEllipse(Pens.Red, new(10, 10, size, size));

            float y = 210;
            for (int month = 3; month <= 5; month++)
            {
                var text =
System.Globalization.CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(month);
                var textSize = e.DrawingContext.MeasureText(text, font);
                e.DrawingContext.DrawText(text, font, Brushes.Black, new(0, y,
textSize.Width, textSize.Height));
                y += textSize.Height + 10;
            }
        }
    }
}
```

13. Build and run the application. The displayed window will look like the following:



Congratulations, you have completed the Rendering Graphics with DrawingContext tutorial.

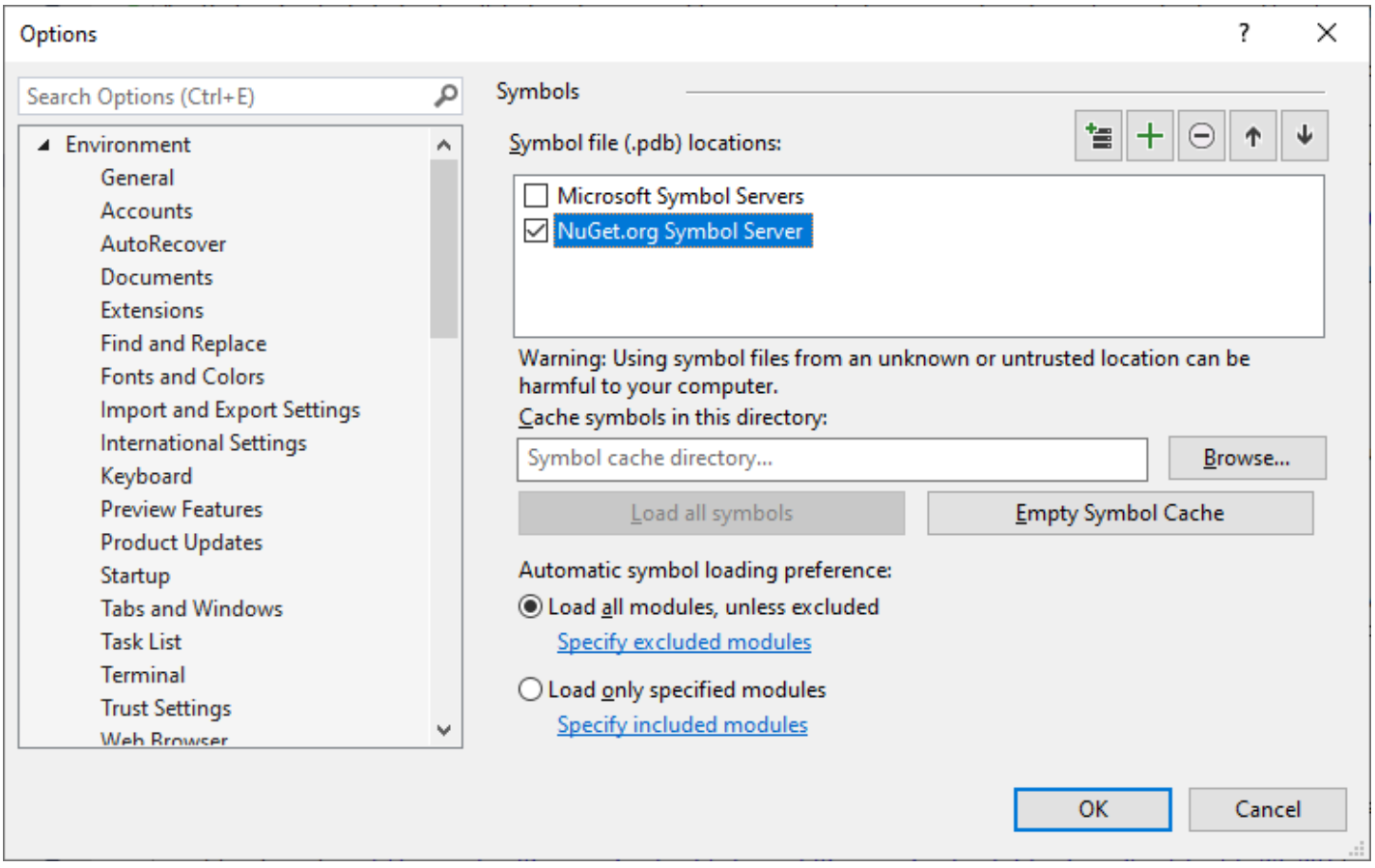
Debugging with AlterNET UI Sources

While debugging your AlterNET UI application, it may be helpful to *step into* the AlterNET UI source code to better understand what is happening under the hood.

Here are the steps required to get AlterNET UI Source debugging working in Visual Studio:

1. Clone or download the AlterNET UI repository on GitHub. Make sure you are using the source tagged with the same version as the version of AlterNET NuGet packages you are using in your project.

2. In Visual Studio, open **Tools > Options > Debugging > Symbols** (or **Debug > Options > Symbols**). Under **Symbol file (.pdb) locations**, check **NuGet.org Symbol Server**.



3. In Visual Studio, open **Tools > Options > Debugging > General** (or **Debug > Options > General**). Ensure **Enable Just My Code** is unchecked.

4. Start debugging your AlterNET UI application. To ensure the symbols are loaded, open **Debug > Windows > Modules**, locate the `Alternet.UI.dll` and other Alternet dlls rows, and select **Load Symbols** in the context menu.

5. After that, executing the **Step Into** command on an AlterNET UI method call will lead to an *open file dialog* prompting to locate an AlterNET UI source code file. In that dialog, navigate to the AlterNET

UI sources directory you prepared in step 1 and find and select the requested source file there.
6. After the initial setup described in the steps above is done, the AlterNET UI sources will be debugged automatically without any additional actions required.

For additional information on consuming NuGet debug symbols, see this MSDN article.

Using AlterNET UI NuGet Packages

The easiest way to create an AlterNET UI application project is to use the project templates that come with AlterNET UI Visual Studio extension or can be installed separately using command line.

However, AlterNET UI Nuget Packages can be referenced manually in a .NET project. To reference AlterNET UI framework in your project, add the following lines to your `.csproj` file:

```
<ItemGroup>
    <PackageReference Include="Alternet.UI" Version="0.9.200-beta" />
</ItemGroup>
```

The `Version` value can be set to one of the published versions of the AlterNET UI packages.

After you have added the packages, you can start a GUI application in your code. You can use the following code to show an empty window in a .NET console application:

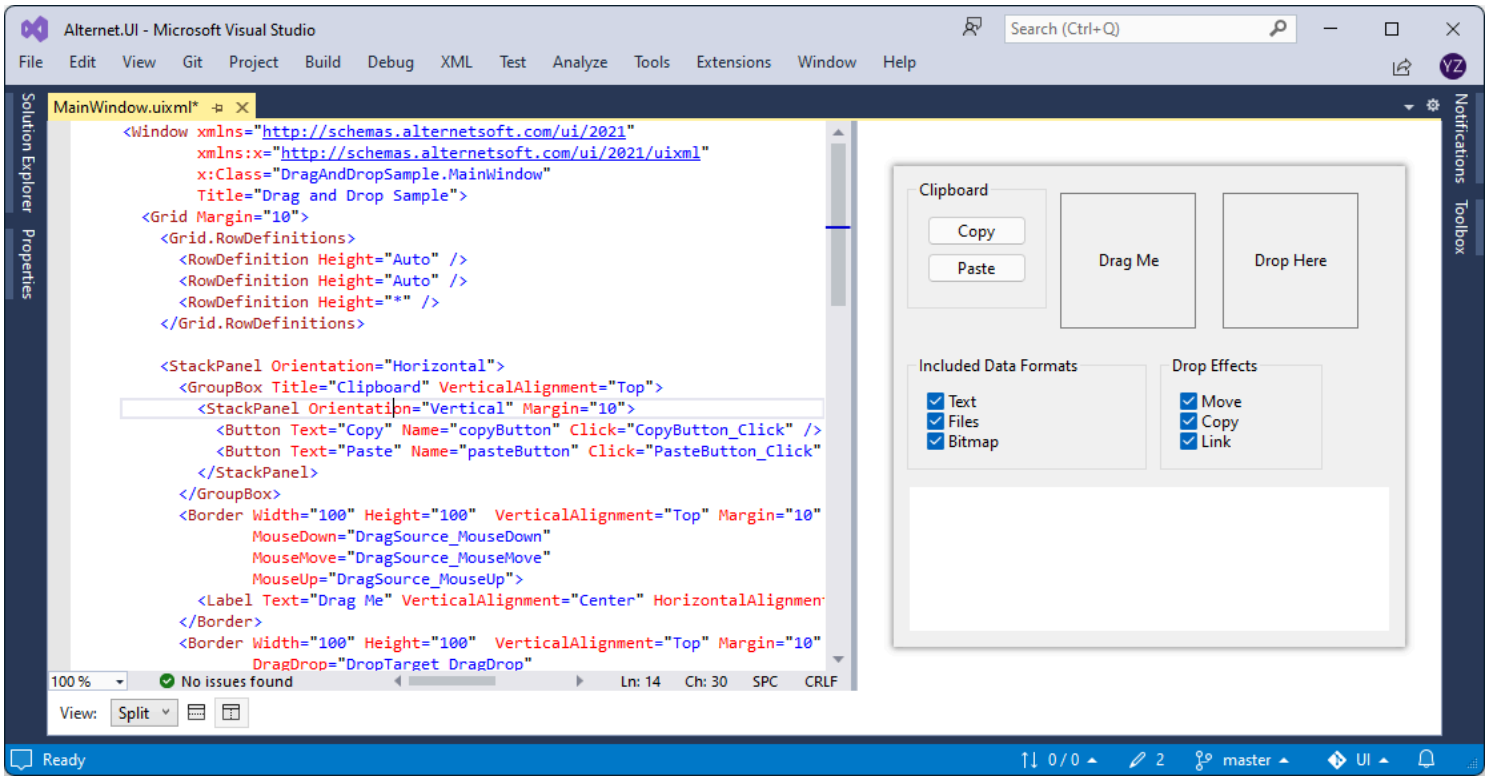
```
class Program
{
    [STAThread]
    public static void Main(string[] args)
    {
        var application = new Alternet.UI.Application();
        var window = new Alternet.UI.Window();

        application.Run(window);

        window.Dispose();
        application.Dispose();
    }
}
```

If you would like to hide the console created by the console application, change its `OutputType` to `WinExe` like so:

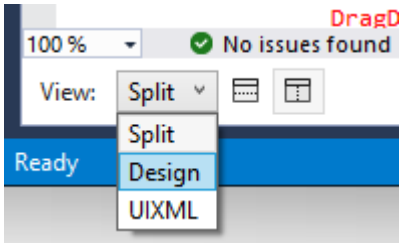
```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
  </PropertyGroup>
```

Using UIXML Previewer in Visual Studio

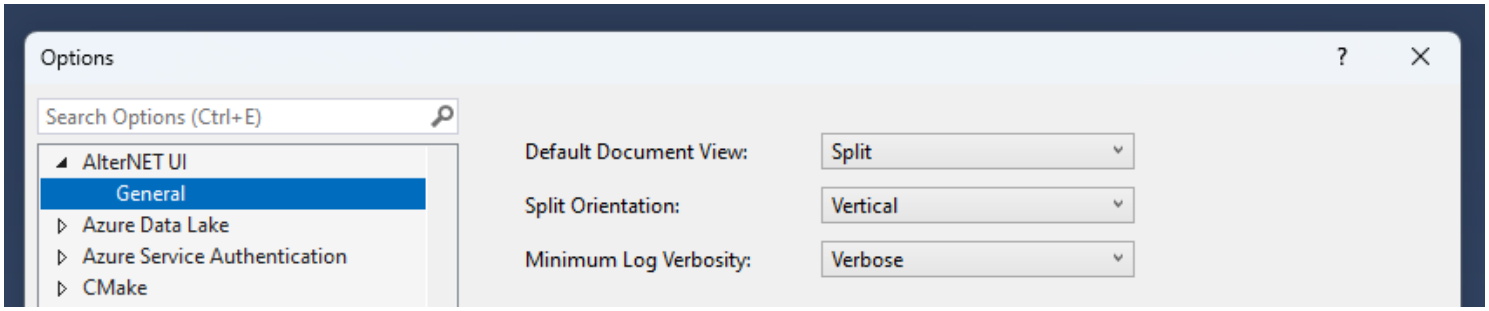
AlterNET UI extension for Visual Studio provides the UIXML previewer feature. It allows for editing UIXML side-by-side with a live preview of the UI being edited.



Several UIXML previewer modes are available: *Split Horizontal*, *Split Vertical*, *Design* (only UIXML preview visible on screen), *UIXML* (only UIXML code editor visible on screen):



You can also set options for the UIXML previewer in the AlterNET UI Extension Options page (click *Tools/Options...* menu item, select AlterNET UI section):



Using SkiaSharp with AlterNET UI

There are different ways to use SkiaSharp:

- You can write an explicit conversions for SKBitmap from/to Image and GenericImage.

For example: `var image = (Image) skBitmap;`.

- You can call Image.LockSurface and GenericImage.LockSurface in order to get SKCanvas.

For example: `using var canvasLock = image.LockSurface(); var canvas = canvasLock.Canvas;`.

Here you can find examples on using SkiaSharp with AlterNET UI.

Example of drawing on PictureBox control

```
        private void Draw(Action<SKCanvas,int,int> action)
        {
            RectI rect = (0, 0, PixelFromDip(pictureBox.Width),
PixelFromDip(pictureBox.Height));

            SKBitmap bitmap = new(rect.Width, rect.Height);

            SKCanvas canvas = new(bitmap);

            canvas.Clear(Color.White);

            action(canvas, rect.Width, rect.Height);

            var image = (Image)bitmap;
            pictureBox.Image = image;
        }
```

GenericImage to SKBitmap and to PictureBox

```
        private void GenericToSkia()
        {
            // Creates generic image from the specified url
            GenericImage image = new(backgroundUrl1);

            // Converts created generic image to SKBitmap
            var bitmap = (SKBitmap)image;

            // Converts SKBitmap to Image and assigns it to PictureBox control
```

```
            pictureBox.Image = (Image)bitmap;
        }
```

Paint UserControl on SKBitmap

```
        private void PaintOnCanvas()
        {
            RectD rectDip = (0, 0, control.Width, control.Height);
            RectI rect = rectDip.PixelFromDip();

            SKBitmap bitmap = new(rect.Width, rect.Height);

            SKCanvas canvas = new(bitmap);
            canvas.Scale((float)control.ScaleFactor);

            canvas.Clear(Color.White);

            SkiaGraphics graphics = new(canvas);

            PaintEventArgs e = new(graphics, rectDip);

            control.RaisePaint(e);

            pictureBox.Image = (Image)bitmap;
        }
```

LockSurface on GenericImage

These methods assume to be members of a Control.

```
        private void LockSurfaceOnGenericImage(bool hasAlpha)
        {
            var width = 700;
            var height = 500;

            var image = GenericImage.Create(PixelFromDip(width),
PixelFromDip(height), Color.Aquamarine);
            image.HasAlpha = hasAlpha;

            using (var canvasLock = image.LockSurface())
            {
                var canvas = canvasLock.Canvas;
                canvas.Scale((float)ScaleFactor);
```

```
            canvas.Clear(Color.White);

            var font = Font.Default;

            canvas.DrawText("Hello", (600, 0), font, Color.Black, Color.LightGreen);

            canvas.DrawRect(SKRect.Create(width, height), Color.Red.AsPen);

            DrawBeziersPoint(canvas);
            canvas.Flush();
        }

        pictureBox.Image = (Image)image;
    }

    private void DrawBeziersPoint(SKCanvas dc)
    {
        Pen blackPen = Color.Black.GetAsPen(3);

        PointD start = new(100, 100);
        PointD control1 = new(200, 10);
        PointD control2 = new(350, 50);
        PointD end1 = new(500, 100);
        PointD control3 = new(600, 150);
        PointD control4 = new(650, 250);
        PointD end2 = new(500, 300);

        PointD[] bezierPoints =
        {
             start, control1, control2, end1,
             control3, control4, end2
         };

        dc.DrawBeziers(blackPen, bezierPoints);
    }
```

LockSurface on Bitmap and draw text

This example method assumes to be a member of a Control.

```
    private void DrawTextOnSkia(bool hasAlpha)
    {
        string s1 = "He|l lo";
        string s2 = "; hello ";
```

```
    var width = 700;
    var height = 500;

    bitmap ??= new Bitmap(PixelFromDip(width), PixelFromDip(height));
    bitmap.HasAlpha = hasAlpha;
    bitmap.SetDPI(GetDPI());

    using (var canvasLock = bitmap.LockSurface())
    {
        var canvas = canvasLock.Canvas;
        canvas.Scale((float)ScaleFactor);

        canvas.Clear(prm.BackColor);
        canvas.DrawRect(SKRect.Create(width, height), Color.Red.AsPen);

        PointD pt = new(10, 10);
        PointD pt2 = new(10, 150);

        var font = Font.Default;

        canvas.DrawText("Hello", (600,0), font, Color.Black, Color.LightGreen);

        canvas.DrawText(s1, pt, font, Color.Black, Color.LightGreen);

        canvas.DrawText(s2, pt2, font, Color.Red, Color.LightGreen);

        canvas.DrawPoint(pt, Color.Red);
        canvas.DrawPoint(pt2, Color.Red);

        DrawBeziersPoint(canvas);

        canvas.Flush();
    }

    pictureBox.Image = bitmap;
}
```