Version 1.0 - May 2022

Getting Started with Python GUI

Using DelphiFMX and DelphiVCL for Python GUI Development

Jim McKeeth **Priyatham**

embarcadero®

PythonGUI.org

Table of Contents

Getting Started with Python GUI:
Using Delphi FMX and VCL for Python
PythonGUI.org

Version 1.0 - May 19, 2022

www.embarcadero.com

About Embarcadero Technologies

– – –

Introduction

Welcome to Python GUI development with DelphiFMX and DelphiVCL. These libraries are perfect for building native apps for Windows and multiple other platforms with Python. The libraries are designed for Python developers, and therefore knowledge of Delphi is not required.

This guide provides an overview of the Delphi4Python architecture and platforms, and walks the reader though installing and using these two libraries:

- DelphiFMX for Python
- DelphiVCL for Python

We also provide demos and code because that's always the fun part. Finally, it will explore what more is possible by mixing Delphi and Python.

Who Should Read This eBook?

The DelphiFMX and DelphiVCL libraries for Python are powered by Delphi, a programming language and toolset by Embarcadero. However, using the libraries does not require any familiarity or experience with Delphi, and is intended for easy use by Python developers. If you work with Python and want to create great user interfaces, then this guide is for you.

The guide, however, is also useful for Delphi developers who want an introduction to Python development as well. It is our hope that this guide and the DelphiVCL and DelphiFMX libraries will help you create amazing graphical user interfaces for your Python projects.

This eBook is a living document, and we will continue to update it, so be sure to check PythonGUI.org for the latest version. While you are there you may discover some other great resources to aid in your Python GUI development efforts

Python vs Delphi

The eBook focuses on programming with Python, but also discusses Delphi as the source for the libraries. It isn't necessary to choose one over the other. Both are good programming languages, and there is room in a programmer's toolbelt for many programming languages giving you the option to use the right tool for the job. So let's explore the possibilities of Unlocked when using these tools together!

Python remains a very popular programming language for new developers and in many specific use cases. The significant characteristics of Python include:

- significantly simpler syntax,
- feels like reading codified English, and
- automatically handles memory leaks.

What is Delphi for Python?

Delphi for Python's primary focus is to provide free Python modules or bindings of Delphi's GUI frameworks to Python developers. There are two different libraries based on Delphi's two different frameworks: DelphiVCL, the native library for Windows; and DelphiFMX, for multiplatform development. This guide focuses on getting you started with both of them.

DelphiVCL for Python

Windows 32-bit and 64-bit only

- Windows 8.1 through Windows 11
- Earlier versions may work but are not supported

Based on native Windows components

Includes Windows Handles, Messages, Accessibility, etc.

High-DPI & Custom Styling

DelphiFMX for Python

Multi-platform for Windows, Linux, Android, and macOS

- Uses GPU for custom rendering
- Supports most common versions

Higher level of abstraction

Platform services simplify behaviors

High-DPI & Custom Styling

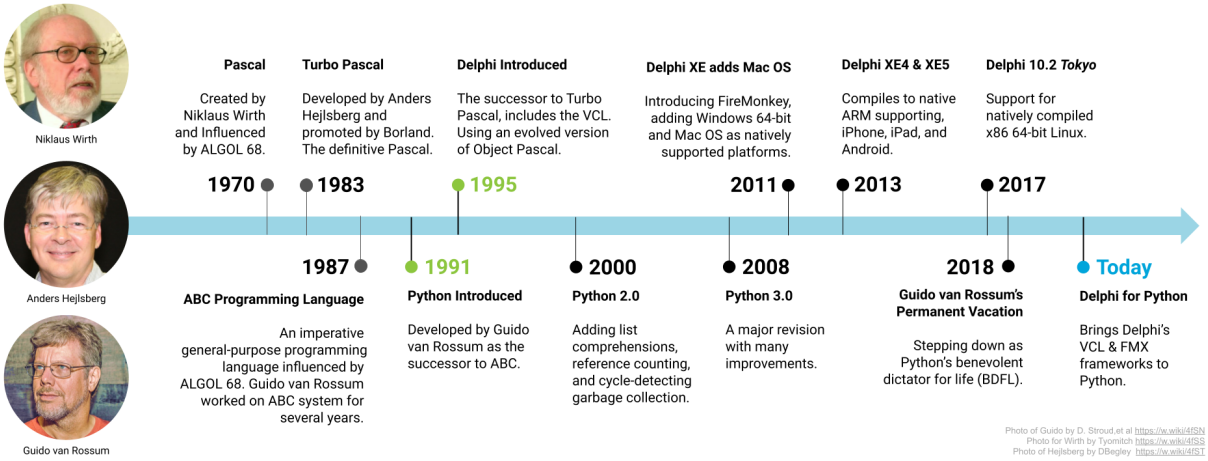
Both libraries are powered by the Python for Delphi[1] library, the same technology that powers the popular PyScripter[2] Python IDE. Python for Delphi is a bi-directional bridge between Python and Delphi, giving Delphi developers access to everything in the world of Python and bringing Delphi libraries to Python developers too. Both libraries are available today on GitHub and PIP, the Python package manager.

[1] *python4delphi*. [online] GitHub. https://github.com/Embarcadero/python4delphi [Accessed 11 May 2022].

[2] *Download PyScripter Free*. https://www.embarcadero.com/free-tools/pyscripter/free-download [Accessed 11 May 2022].

We support both legacy, prebuilt Python virtual environments and Conda environments. While the libraries are under active development, they are ready to use today. Go ahead and start using them, create projects, and file issues on GitHub. We are open to all suggestions.

A Timeline for Delphi and Python



In 1970, Nicklaus Wirth created the Pascal program language based on or influenced by ALGOL 68. It was designed to teach good programming practices, structured programming, and more. In 1983, Anders Hejlsberg created Turbo Pascal, which was initially called Blue Label Pascal. The Borland Software Corporation acquired it, hired Hejlsberg, and promoted the Pascal language. Several Pascal dialects evolved from the original. Then in 1987, the ABC programming language was born.

ABC was an imperative general-purpose programming language on which Guido van Rossum worked for a while. He went on in 1991 to make Python, the successor to ABC. In 1995, Delphi, a dialect of Object Pascal, was introduced as the successor to Pascal. This first release included a graphical user interface (GUI) framework VCL along with Delphi. Quite a few languages came out in 1995. In fact, Delphi and Python are about the same age. Later, Python 2.0 and Python 3.0 were released, which moved the Python language forward. And then, in 2011, Delphi introduced Mac support with the FireMonkey GUI framework for cross-platform support. And in 2013, support was given to the ARM compiler with Delphi, adding iOS and Android support.

In 2017, Embarcadero released Delphi Tokyo, which brought Linux support. Then, in 2018 Guido van Rossum stepped down and took a permanent vacation as Python's "benevolent dictator" for life. And that brings us to today, where we're going to talk about Delphi for Python.

Delphi's DNA Compared to the Zen of Python

One of the essential considerations of Delphi is developer productivity. To help them achieve productivity, it provides:

- A WYSIWYG visual designer to drag and drop components
- A wide range of both visual and nonvisual components and the properties and events associated with them
- Backward compatibility to older versions of Delphi
- A rich component ecosystem

Developers can quickly make fast, powerful applications using the above.

We want to compare Delphi's DNA with the Zen of Python[3], which has 19 guiding principles that influence the design of the Python programming language. Let's check out a few of those:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.

So in a lot of ways, Delphi and Python share the same philosophies.

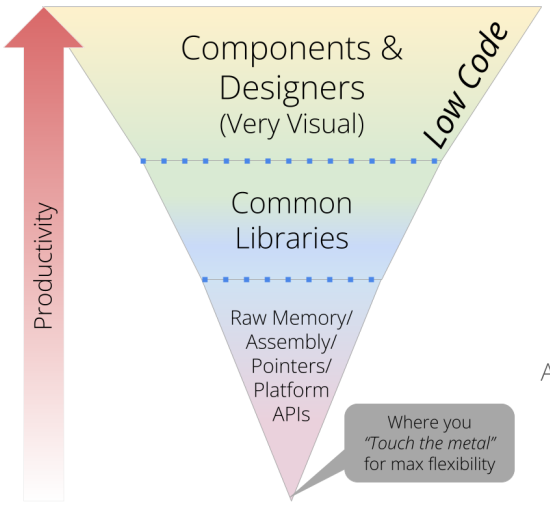
What are the differences between Delphi and Python?

Some key differences are these:

- Although commercial licenses are available, Delphi also has a free community edition, whereas Python is open source.
- Python has garbage collection with reference counting and cycle detection. Delphi has manual memory management, but it has an ownership model that simplifies it for most scenarios.
- Delphi is a compiled language, although there are some third-party interpretation options. In contrast, Python is primarily interpreted with some options to compile it.
- People build for business purposes, business-to-business type software, or internal IT development using Delphi. Although Python has grown as a general-purpose language, it's prevalent in research and prototyping machine learning.

[3] Peters, T., 2004. *PEP 20 – The Zen of Python | peps.python.org*. [online] Python Software Foundation. https://peps.python.org/pep-0020/ [Accessed 10 May 2022].

Which Levels of Development are Home to Delphi and Python?



Delphi makes it easy to mix *all three*!

The productivity of components with the flexibility & power of Platform APIs.

Python has tremendous power at *second and base levels*!

Python has extensive range of common libraries. Also, can access raw memory using Python's C API.

Development, using Delphi or in general, falls into one of these three categories:

1. Low code, with visual components and designers
2. Standard and third-party libraries to implement specific functionalities
3. Raw memory access using pointers or assembly code

Delphi has many low-code tools designed for people who don't want to learn programming and just want to get something done. Those tools are very visual, so very little code is written and the focus is on productivity. There are limitations on what you can do at this low-code level. At the second level, we use common libraries and write programs. Here we can customize more specific functionality compared to level one. Then you have the lowest level, which is the most flexible and most powerful but the least productive. Here, you're dealing with raw memory and pointers by writing assembly language code. You're talking directly to the CPU with archaic platforms and APIs.

Delphi does an excellent job of combining all three levels of development. Python mainly falls into the middle level, with vast built-in and third-party libraries. Python has automatic garbage collectors, whereas Delphi deals with garbage manually. We can extend Python[4] with the C language. Many third-party Python modules are written in C or C++, which gives you raw memory management and extends Python's reach to low-level platform access.

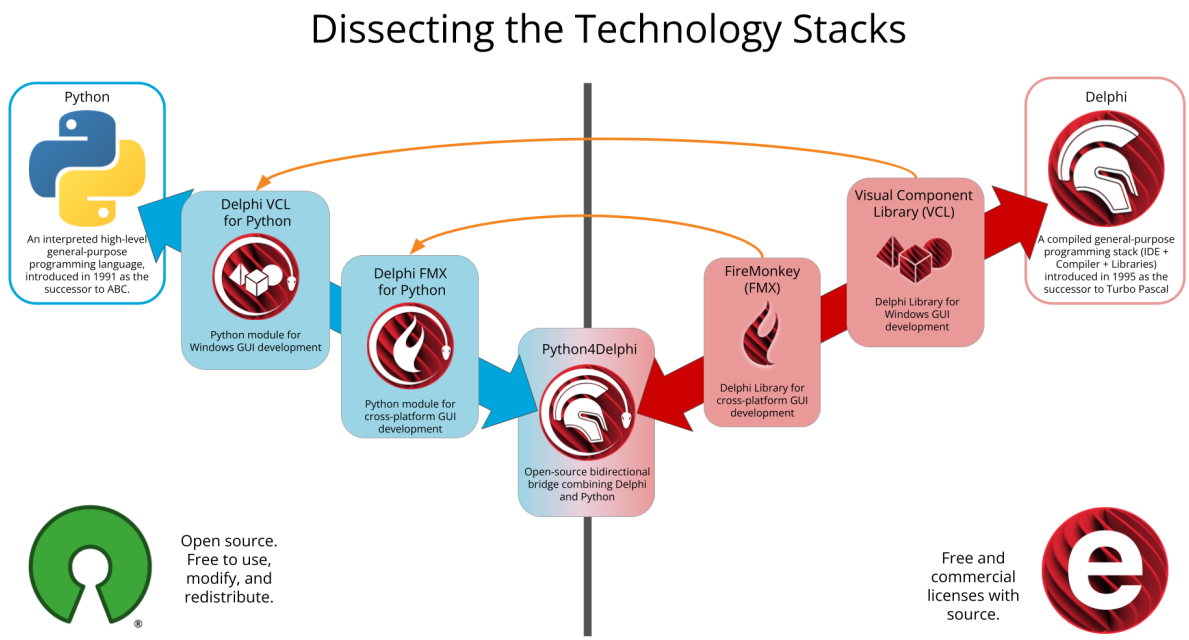
[4] *Extending Python with C or C++* [online] https://docs.python.org/3/extending/extending.html [Accessed 13 May 2022].

Python's provision to create extension modules in C and offer Python bindings has made it the most used language. Furthermore, we can import those extension modules as libraries, making Python incredibly productive. This is our primary focus area, where we also created the extension modules for Python that are written in Delphi.

Delphi's VCL and FireMonkey Libraries

Delphi offers two libraries or frameworks for graphical user interface (GUI) application development:

- Visual Component Library (VCL)
- FireMonkey (FMX)



VCL was initially released along with the first Delphi back in 1995, focused on Windows. Even though Delphi is not open source, it shipped with source code, making a huge difference in its adoption. Delphi is built with the component model for component reuse. It gave all the developers access to the source code to see how Delphi did that and create their custom components. So it created a rich ecosystem of components for developers to use. VCL is based on Windows components, and all of them have handles to intercept messages. A significant part of the VCL framework wraps standard Windows components. So you have automatic access to everything that comes with a Windows component, but it adds a layer of abstraction. VCL simplifies their usage, and you don't have to think about handles, messages, or other things.

The **FireMonkey** framework was released in 2011, as you can see in the timeline above. FireMonkey is designed from the ground up as a cross-platform GUI framework. It takes advantage of GPUs. So hardware acceleration uses DirectX on Windows and OpenGL on other platforms to create very fast, great-looking UIs across platforms. It supports Windows, Mac, iOS, Android, and Linux. If you know FireMonkey, you can shift to VCL quickly, but FireMonkey is not hindered by trying to be completely backward compatible with the VCL. It includes platform services, which help with the abstraction while moving across platforms. As a result the behaviors and the look and feel automatically adapt to the platform you're running on.

With the recent advancements, Delphi is as fast as C++. We created Python extension modules for VCL and FireMonkey using the Python for Delphi[5] bridge. So these two libraries make up Delphi for Python, essentially creating:

- DelphiVCL for Python[6]
- DelphiFMX for Python[7]

Getting Started with Both DelphiVCL for Python and DelphiFMX for Python

DelphiVCL for Python is a Python binding of Delphi's native Windows GUI framework-VCL, whereas DelphiFMX for Python is a binding of a cross-platform GUI framework—FireMonkey (FMX). The FMX framework supports the most popular operating system platform but VCL is built specifically for Windows. You don't need to install or know Delphi to develop GUI applications using both DelphiFMX and DelphiVCL. DelphiFMX provides common functionality on top of platform-native APIs, whereas DelphiVCL takes advantage of native Windows controls. The only requirement is installing the **`delphifmx`** and **`delphivcl`** packages in the Python environment.

Much of what you learn with either DelphiVCL[6] or DelphiFMX[7] will also apply to the other package. While FMX supports Windows as well, you might find some use cases in which you prefer the lighter-weight VCL framework when creating Windows applications.

Installation

Both package distributions are available via PyPi[8] or by downloading the source via GitHub. DelphiFMX is compiled for Android32, Android64, Linux64, OSX64, OSXARM64, Win32, and Win64 and has been tested and working on all the mentioned platforms. DelphiVCL is compiled only for Windows. All Python versions from 3.6 to 3.10 are supported, including Conda.

[5] *Embarcadero's Fork of Python4Delphi*. https://github.com/Embarcadero/python4delphi
[6] *DelphiVCL for Python. GitHub.* https://github.com/Embarcadero/DelphiVCL4Python
[7] *DelphiFMX for Python. GitHub.* https://github.com/Embarcadero/DelphiFMX4Python
[8] *PyPi's page for Embarcadero.* https://pypi.org/user/Embarcadero/

The easiest way to install is via PIP:

DelphiFMX:

```
pip install delphifmx
```

DelphiVCL:

```
pip install delphivcl
```

You can also install manually by downloading or cloning the repository from GitHub: github.com/Embarcadero/DelphiFMX4Python or github.com/Embarcadero/DelphiVCL4Python. After cloning or downloading, enter the root **DelphiFMX4Python** or **DelphiVCL4Python** folder/directory and open the command prompt or Anaconda prompt with that path. Now install the package using:

```
python setup.py install
```

Testing the Installation

Note: The REPL codes written below for **`delphivcl`** are on a Windows-based computer. The REPL codes for **`delphifmx`** can be written on any computer based on Windows/Linux/macOS operating systems. For the purpose of this tutorial, let's assume we're on a Windows-based computer because it supports both **`delphivcl`** and **`delphifmx`** package distributions.

After installing the packages using **`pip`**, let's enter the Python REPL to understand a few essential things. Python has a predefined **`dir()`** function that lists available names in the local scope. So, before importing anything, let's check the available names using the **`dir()`** function.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

Now let's import the installed **`delphifmx`** module to validate its installation and check for the output of the **`dir()`** function:

```
>>> import delphifmx
>>> import delphivcl
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'delphifmx', 'delphivcl']
```

In the above output list, we have **delphifmx** and **delphivcl** as part of the available names in the local scope. In this case, if we need to use any classes or functions available in the **delphifmx** module, we should use a dot (.) operator after it.

```
>>> dir(delphifmx)[0:45]
['Abort', 'Action', 'ActionList', 'AdapterListView', 'AniIndicator',
'AppearanceListView', 'Application', 'Arc', 'ArcDial',
'BaseBindScopeComponent', 'BaseLinkingBindSource', 'BaseObjectBindSource',
'BasicAction', 'BasicBindComponent', 'BindComponentDelegate',
'BindingsList', 'BitmapTrackBar', 'Bounds', 'Button', 'CalloutPanel',
'CalloutRectangle', 'CameraComponent', 'CaretRectangle', 'CheckBox',
'CheckColumn', 'Circle', 'Collection', 'ColorBox', 'ColorButton',
'ColorComboBox', 'ColorListBox', 'ColorPanel', 'ColorPicker', 'ColorQuad',
'Column', 'ComboBox', 'ComboColorBox', 'ComboEdit', 'ComboEditBase',
'CommonCustomForm', 'Component', 'ContainedAction', 'ContainedActionList',
'ContainedBindComponent', 'Control']
>>> dir(delphivcl)[0:45]
['Abort', 'Action', 'ActionList', 'ActivityIndicator', 'Application',
'BasicAction', 'Bevel', 'BitBtn', 'Bitmap', 'BoundLabel', 'Button',
'Canvas', 'CheckBox', 'Collection', 'ColorBox', 'ComboBox', 'Component',
'ContainedAction', 'ContainedActionList', 'Control', 'ControlBar',
'CreateComponent', 'CustomAction', 'CustomActionList',
'CustomActivityIndicator', 'CustomControl', 'CustomDrawGrid', 'CustomEdit',
'CustomForm', 'CustomGrid', 'CustomMemo', 'CustomStyleServices',
'CustomTabControl', 'CustomToggleSwitch', 'DateTimePicker',
'DelphiDefaultContainer', 'DelphiDefaultIterator', 'DelphiMethod',
'DrawGrid', 'Edit', 'FileOpenDialog', 'Form', 'FreeConsole', 'Graphic',
'GroupBox']
```

To avoid an enormous list of names, we checked for the first 46 elements only using **dir(delphifmx)[0:45]** and **dir(delphivcl)[0:45]**. Let's check for a few available classes, functions, and objects. Both packages have the respective names in their scope as you can observe from the above list of available names.

```
>>> delphifmx.CreateComponent
<built-in function CreateComponent>
>>> delphivcl.CreateComponent
<built-in function CreateComponent>
>>> delphifmx.Button
<class 'Button'>
>>> delphivcl.Button
<class 'Button'>
>>> delphifmx.Form
<class 'Form'>
```

```
>>> delphivcl.Form
<class 'Form'>
>>> delphifmx.Application
<Delphi object of type TApplication at 557897D1E6A0>
>>> delphivcl.Application
<Delphi object of type TApplication at 557897D1F5C2>
```

We need to create instances/objects for classes like **Button** and **Form**. There are many other classes and functions but only one object **Application** instance, which is an existing singleton instance, ready for use with the dot (.) operator. **Application** is the source of all GUI applications that we create.

VCL Guide 1: An Empty and Simplest App

Let's now create a simple GUI application. The code for it is:

```
from delphivcl import *

Application.Initialize()
Application.Title = "Hello Delphi VCL"
app = Form(Application)
app.SetProps(Caption = "Welcome")
app.Show()
FreeConsole()
Application.Run()
app.Destroy()
```

Using the above code, we just created an empty GUI app. Please save the above code and run it to see the following output:

Let's explore and understand the functionality of the code.

```
from delphivcl import *

Application.Initialize()
Application.Title = "Hello Delphi VCL"
```

At first, we imported everything from **delphivcl**. Then, we initialized the application and set a title for it. Later, we will create the GUI application window using the following code:

```
app = Form(Application)
app.SetProps(Caption = "Welcome")
```

We can refer to all the classes that are part of the import as components. The **Form** is special and different as it creates the GUI window containing all other components. We instantiated the **Form** with **Application** as the owner parameter in the above code. All the components, including **Form**, have a method **setProps()** to set their properties. Here we've set **Caption** that appears on the title bar of the Form window.

Let's look at the following few lines of the code:

```
app.Show()
```

```
FreeConsole()
Application.Run()
app.Destroy()
```

As we created the application and set its properties, we will show it on the screen using the **app.show()** code snippet. GUI applications run in interaction with the command window (console). To make the GUI perform better without lags, we use **FreeConsole()** to give primary control to the GUI interface. **Application.Run()** starts the GUI interaction loop between the GUI and the user of the GUI application. When we close the GUI application, **app.Destroy()** takes care of not crashing it.

FMX Guide 1: An Empty and Simplest App

Let's now create an FMX version of the above-created simplest VCL application. The code for it is:

```
from delphifmx import *

Application.Initialize()
Application.Title = "Hello DelphiFMX"
app = Form(Application)
app.SetProps(Caption = "Welcome")
Application.MainForm = app
app.Show()
Application.Run()
app.Destroy()
```

The differences between the above VCL version of the Simplest app and this FMX code are as follows:

- Here, we've set the **Application.MainForm** to the **Form** object. When we create multiple GUI windows, we need one main form/window to control others. The MainForm is set automatically to the first form/window behind the scenes for us in VCL.
- We didn't use **FreeConsole()** here because it happens automatically in DelphiFMX.

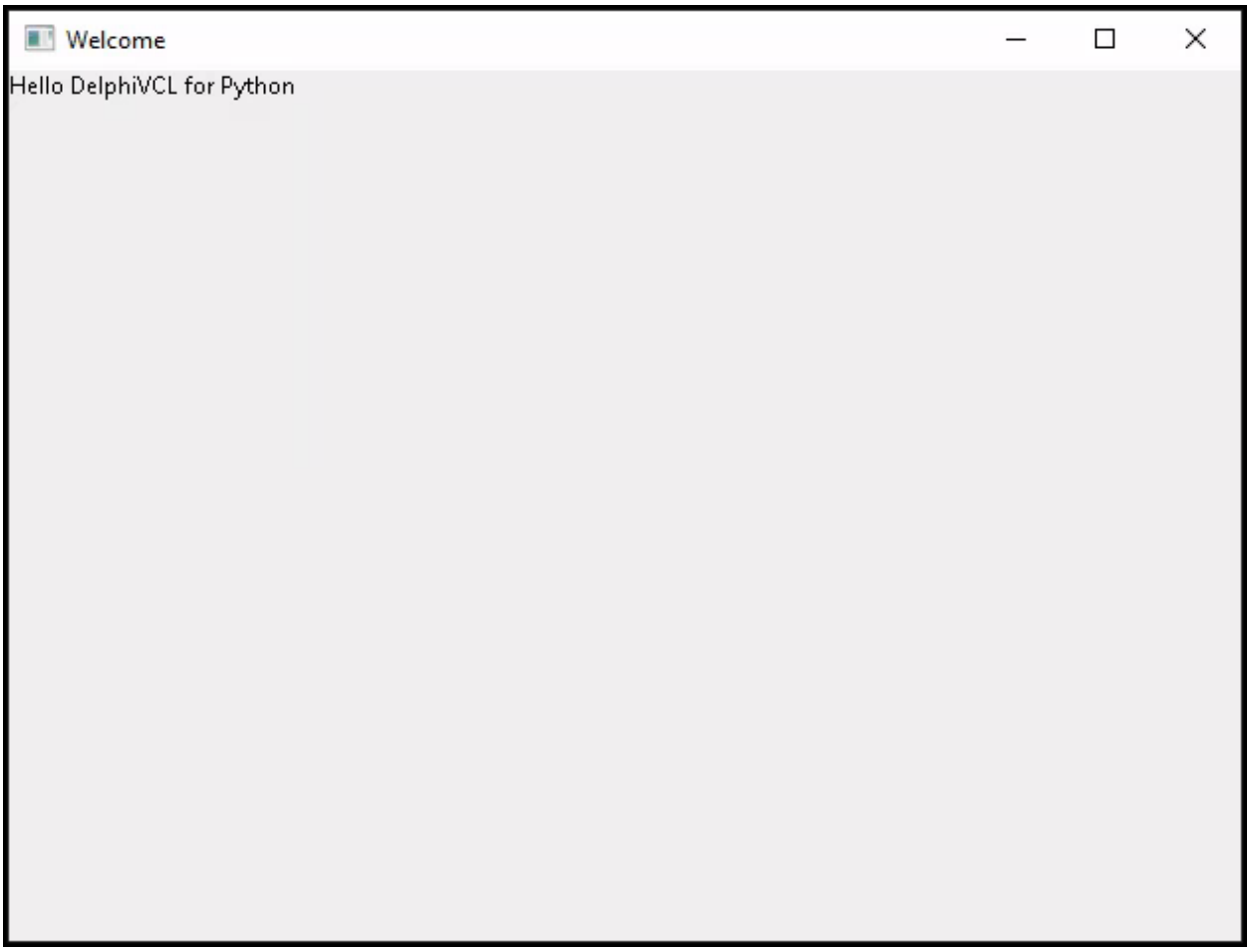
VCL Guide 2: Hello, DelphiVCL for Python

We discussed the most basic ideas about the **`delphivcl`** library in the first Simplest guide. We created an empty GUI application without displaying anything on the Form/GUI window. Also, we did not use an object-oriented approach to create the GUI application. So let's expand on those ideas and develop an object-oriented version displaying a text message.

First, let's look at the code. You might be able to guess what the below code does if you understood the basics from the first guide.

```
from delphivcl import *

class GUIApp(Form):

    def __init__(self, owner):
        self.SetProps(Caption = "Welcome")

        self.lblHello = Label(self)
        self.lblHello.SetProps(
            Parent=self,
            Caption="Hello DelphiVCL for Python")

def main():
    Application.Initialize()
    Application.Title = "Hello DelphiVCL"
    app = GUIApp(Application)
    app.Show()
    FreeConsole()
    Application.Run()
    app.Destroy()

main()
```

As you save the above code in a Python file and run it, you'll get the following GUI window with a text message as follows:



In the main function, let's look at the following line of code:

```
    app = GUIApp(Application)
```

Instead of instantiating the **Form** directly, we instantiate a class - **GUIApp** that extends the **Form** class. Let's investigate the code in the **GUIApp** class:

```
class GUIApp(Form):

    def __init__(self, owner):
        self.SetProps(Caption = "Welcome")

        self.lblHello = Label(self)
        self.lblHello.SetProps(
            Parent=self,
            Caption="Hello DelphiVCL for Python")
```

As we instantiated the **GUIApp** using **app = GUIApp(Application)**, the **owner** argument was assigned with the **Application** object. After that, **Form** uses the **owner** in its initialization and creates an empty Form/GUI window. This **owner** variable can have any other name as it's just a placeholder of the **Application** object. In the first line of the **GUIApp** initialization, we've set the **Caption** property of the **Form**.

Then we instantiated the **Label** component/class with the instance/object of the **Form** as its parameter using the **self.lblHello = Label(self)** code snippet. We use **Label** to display any single-line text messages. Every component other than **Form** will have a parent and is set using the **Parent** property. The parent holds the child component in it.

In our code, we're setting **Label**'s parent as **Form** using **Parent=self**. So now the **Form** object - **app** holds the **Label** object - **lblHello**. Next, the text of the **Label** is set using its **Caption** property. So the Form/GUI window gets populated by a text message: - **Hello DelphiVCL for Python**.

We used all the default positions and sizes of the **Form** and **Label** and didn't handle any events in this guide. However, we will implement them and introduce some new components in the following advanced guides.

FMX Guide 2: Hello, DelphiFMX for Python

Let's now create an FMX version of the above-created VCL application in Guide 2. The code for it is:

```
from delphifmx import *

class GUIApp(Form):

    def __init__(self, owner):
        self.SetProps(Caption = "Welcome")

        self.lblHello = Label(self)
        self.lblHello.SetProps(
            Parent=self,
            Text="Hello DelphiFMX for Python")

def main():
    Application.Initialize()
    Application.Title = "Hello DelphiFMX"
    app = GUIApp(Application)
    Application.MainForm = app
    app.Show()
    Application.Run()
```

```
    app.Destroy()

main()
```

The difference between the above VCL version of Guide 2 and this FMX code is that:

- In the **Label** component we used **Text** property in place of **Caption** property.

Other than the differences described here and in Guide 1, all the other concepts and explanations are the same.

Note: Along with the above, there are some other differences between VCL and FMX code implementations. We're currently working on a unified API so that we can remove the code differences. Then, one code will work for both libraries.

VCL Guide 3: TODO Task App

We discussed the most basic ideas about the **delphivcl** and **delphifmx** libraries in the first and second guides. We created a GUI application that just displays a simple text on the Form/GUI window.

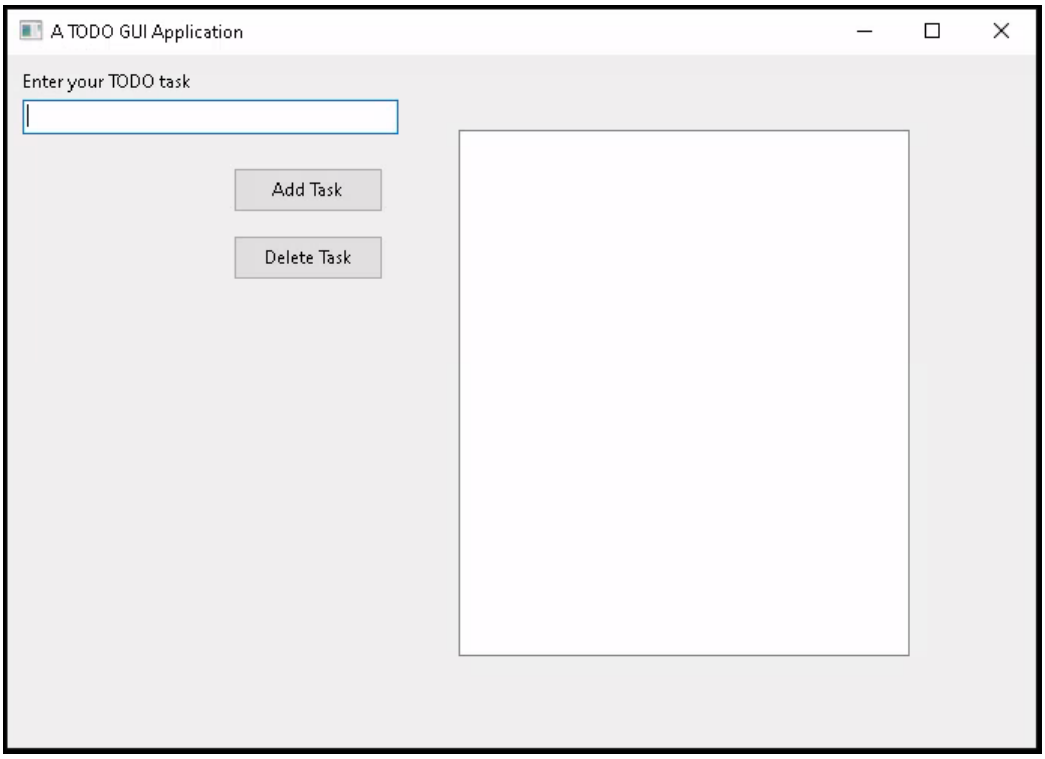
Here we will create a TODO task application to understand some components of GUI applications.

Let's take a look at the code to achieve that:

```
from delphivcl import *

class TodoApp(Form):

    def __init__(self, Owner):
        self.Caption = "A TODO GUI Application"
        self.SetBounds(100, 100, 700, 500)

        self.task_lbl = Label(self)
        self.task_lbl.SetProps(Parent=self, Caption="Enter your TODO task")
        self.task_lbl.SetBounds(10, 10, 125, 25)

        self.task_text_box = Edit(self)
        self.task_text_box.SetProps(Parent=self)
        self.task_text_box.SetBounds(10, 30, 250, 20)
```

```
        self.add_task_btn = Button(self)
        self.add_task_btn.Parent = self
        self.add_task_btn.SetBounds(150, 75, 100, 30)
        self.add_task_btn.Caption = "Add Task"
        self.add_task_btn.OnClick = self.__add_task_on_click

        self.del_task_btn = Button(self)
        self.del_task_btn.SetProps(Parent = self, Caption = "Delete Task")
        self.del_task_btn.SetBounds(150, 120, 100, 30)
        self.del_task_btn.OnClick = self.__del_task_on_click

        self.list_of_tasks = ListBox(self)
        self.list_of_tasks.Parent = self
        self.list_of_tasks.SetBounds(300, 50, 300, 350)

    def __add_task_on_click(self, Sender):
        self.list_of_tasks.Items.Add(self.task_text_box.Text)
        self.task_text_box.Text = ""

    def __del_task_on_click(self, Sender):
        self.list_of_tasks.Items.Delete(0)

def main():
    Application.Initialize()
    Application.Title = "TODO App"
    app = TodoApp(Application)
    app.Show()
    FreeConsole()
    Application.Run()
    app.Destroy()

main()
```

As you save and run the above code, you should get the following GUI as a result:



Let's get to the details of what our code does behind the scenes. First, take a look at the **main()** function:

```
def main():
    Application.Initialize()
    Application.Title = "TODO App"
    app = TodoApp(Application)
    app.Show()
    FreeConsole()
    Application.Run()
    app.Destroy()
```

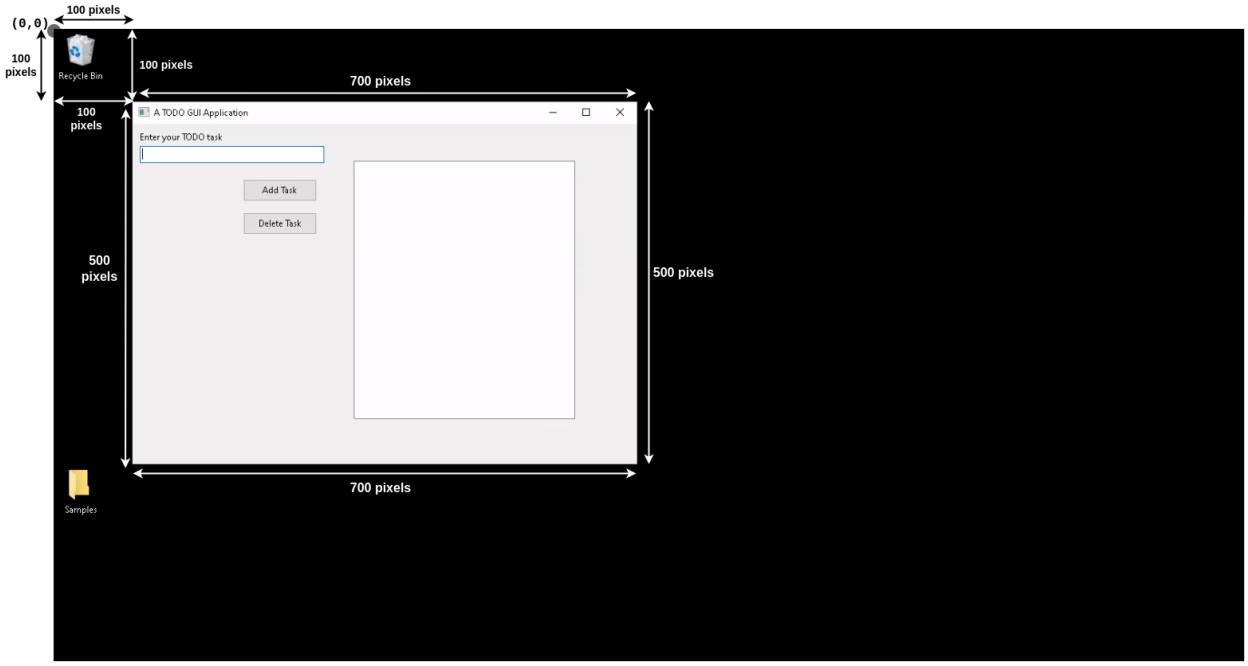
Above, we instantiated the **TodoApp** class with **Application** as the **Owner**. As we instantiate the GUI using **app = TodoApp(Application)**, the following code runs:

```
class TodoApp(Form):

    def __init__(self, Owner):
        self.Caption = "A TODO GUI Application"
        self.SetBounds(100, 100, 700, 500)
```

We inherit the **`Form`** class from the **`delphivcl`** library to create our GUI. In DelphiVCL, all the GUIs are treated as forms. The name of the GUI pop-up window is set using the **`Caption`** property/attribute. The line **`self.SetBounds(100, 100, 700, 500)`** is used to set:

- the GUI window's origin position comparable to the screen's origin position = **`(100, 100);`**
- width of the GUI window = **`700`** pixels; and
- height of the GUI window = **`500`** pixels.

The upper-left corner of the screen is treated as the **`(0, 0)`** coordinate, with the left side as positive width and down as positive height. We can visualize it as shown below:



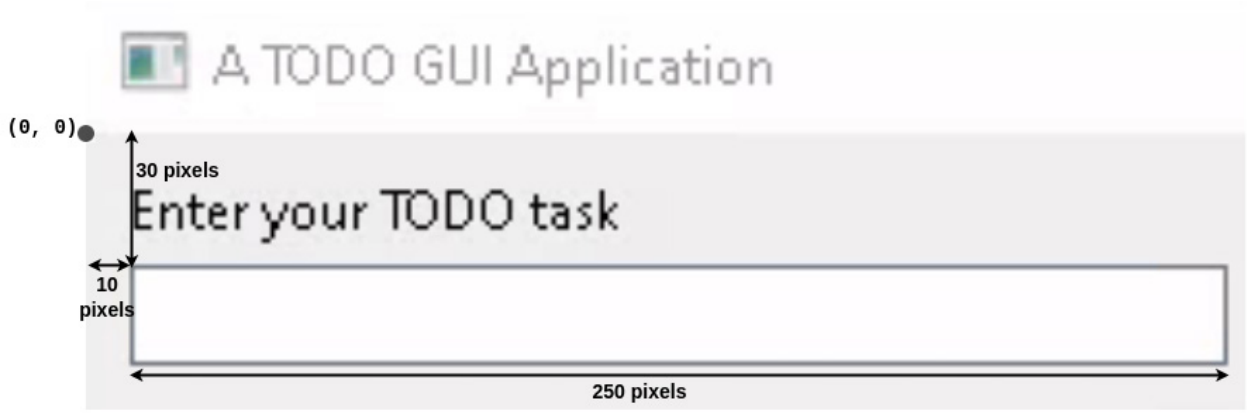
Let's look at the following few lines of code:

```
        self.task_lbl = Label(self)
        self.task_lbl.SetProps(Parent=self, Caption="Enter your TODO task")
        self.task_lbl.SetBounds(10, 10, 125, 25)

        self.task_text_box = Edit(self)
        self.task_text_box.Parent = self
        self.task_text_box.SetBounds(10, 30, 250, 20)
```

The first three lines of code above will create the text “- Enter your TODO task” that you see on the GUI app. It does so by instantiating the **`Label`** class of the **`delphivcl`** library. Every component (**`Label`** here) has a **`SetProps()`** method to set its properties. Every component will have a scope that is set using its **`Parent`** property/attribute, which is set to **`self`** here. The

Caption property sets the string of the text label. Similar to the Form/GUI app, every component needs to be placed inside the Form/GUI using the **SetBounds()** method. For components, the top-left corner of their parent (GUI window here) is considered as the origin - **(0, 0)**.

The next three lines of code create the edit box using the **Edit** class. We can also set the properties/attributes directly without using the **SetProps()** method, like we did here using the code **self.task_text_box.Parent = self**. With the Form/GUI window as the parent of the Edit box, we can visualize its position and size as shown in the below figure. The height of the Edit box is automatically set to the default value.



Let's look at the next few lines of code:

```
        self.add_task_btn = Button(self)
        self.add_task_btn.Parent = self
        self.add_task_btn.SetBounds(150, 75,100,30)
        self.add_task_btn.Caption = "Add Task"
        self.add_task_btn.OnClick = self.__add_task_on_click

        self.del_task_btn = Button(self)
        self.del_task_btn.SetProps(Parent = self, Caption = "Delete Task")
        self.del_task_btn.SetBounds(150,120,100,30)
        self.del_task_btn.OnClick = self.__del_task_on_click
```

The above lines of code create two buttons, **Add Task** and **Delete Task**, using the **Button** instance of the **delphivcl** package. For the buttons, one extra thing you'll find is an event handling using **self.add_task_btn.OnClick = self.__add_task_on_click** and **self.del_task_btn.OnClick = self.__del_task_on_click** for **Add Task** and **Delete Task** buttons, respectively. We will look at this in just a while.

Let's look at the next few lines of code:

```
        self.list_of_tasks = ListBox(self)
```

```
        self.list_of_tasks.Parent = self
        self.list_of_tasks.SetBounds(300,50,300,350)
```

In the above lines of code, we created a list box using the **ListBox** instance and set its **Parent** and position.

Let's now look at the event-handling methods for the **Add Task** and **Delete Task** buttons:

```
    def __add_task_on_click(self, Sender):
        self.list_of_tasks.Items.Add(self.task_text_box.Text)
        self.task_text_box.Text = ""

    def __del_task_on_click(self, Sender):
        self.list_of_tasks.Items.Delete(0)
```

For all the events other than **OnClick**, the Form automatically sends a single argument (**Sender** here, but this can be any name). We can add a task to the list box by typing anything into the text box and pressing the **Add Task** button. Delphi VCL library-based GUIs support tab controls too, where you can also navigate from one component to another using the tab key. So you can press `Tab` on the keyboard, and when the **Add Task** button gets highlighted, you can press `Enter/Return` to fire its event. We add text from the text box to the list box using the **Add()** method under **Items** under **ListBox** instance. We delete the earlier added events on a first-come, first-serve basis by pressing the **Delete Task** button.

FMX Guide 3: TODO Task App

Let's create an FMX version of the VCL "TODO Task App" GUI application.

Let's take a look at the code to achieve that:

```
from delphifmx import *

class TodoApp(Form):

    def __init__(self, Owner):
        self.Caption = "A TODO GUI Application"
        self.SetBounds(100, 100, 700, 500)

        self.task_lbl = Label(self)
        self.task_lbl.SetProps(Parent=self, Text="Enter your TODO task")
        self.task_lbl.SetBounds(10, 10, 150, 25)
```

```
        self.task_text_box = Edit(self)
        self.task_text_box.SetProps(Parent=self)
        self.task_text_box.SetBounds(10, 30, 250, 20)

        self.add_task_btn = Button(self)
        self.add_task_btn.Parent = self
        self.add_task_btn.SetBounds(150, 75, 100, 30)
        self.add_task_btn.Text = "Add Task"
        self.add_task_btn.OnClick = self.__add_task_on_click

        self.del_task_btn = Button(self)
        self.del_task_btn.SetProps(Parent=self, Text="Delete Task")
        self.del_task_btn.SetBounds(150, 120, 100, 30)
        self.del_task_btn.OnClick = self.__del_task_on_click

        self.list_of_tasks = ListBox(self)
        self.list_of_tasks.Parent = self
        self.list_of_tasks.SetBounds(300, 50, 300, 350)

    def __add_task_on_click(self, Sender):
        self.list_of_tasks.Items.Add(self.task_text_box.Text)
        self.task_text_box.Text = ""

    def __del_task_on_click(self, Sender):
        self.list_of_tasks.Items.Delete(0)

def main():
    Application.Initialize()
    Application.Title = "TODO App"
    app = TodoApp(Application)
    Application.MainForm = app
    app.Show()
    Application.Run()
    app.Destroy()

main()
```

The difference between the above VCL version of Guide 3 and this FMX code is that:

- In the **Button** component we used **Text** property in place of **Caption** property.

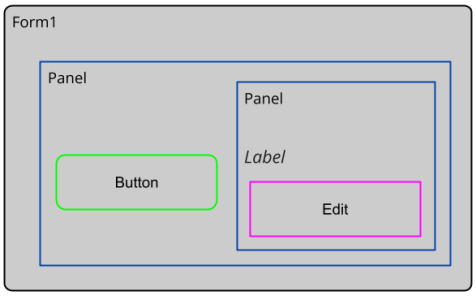
Delphi has an ownership model, where all components can have an owner. Every component has an owner and a parent. In the above sample script, **`Application`** is the owner of the form, and this works differently with the **`Parent`**.

Owner
(Lifecycle)

- Application
  - Form1
    - Panel
    - Label
    - Edit
    - Button
    - Panel
    - Components
  - Form2
    - Components

Parent
(Drawing)

- Form1
  - Panel
    - Button
    - Panel
      - Label
      - Edit
- Form2
  - Nested Components
    - Nested Components
      - Nested Components



All visual components (except the **`Form`**) must have a **`Parent`** defined. The parent is used when drawing the components. A component is drawn on its parent. Both visual and non-visual components have an **`Owner`** property, but the value is optional. The owner is used for object life cycle and memory management. The application owns the forms, and then the forms own the components on it.

In our example, we have a single **`Form`**, but using the DelphiFMX library, you can create multiple forms too. If you have multiple forms, each form owns all of its components, and then those forms are owned by the **`Application`** object. The parent, on the other hand, is associated with the visual scope of the components. A **`Panel`** is a composite component in that it can act as a parent for other components inside it.

The parent can also be the owner, but that isn't required. From the above image, we understand the illustrated concepts of Application Life Cycle and Owner-Parent-Child interaction.

Introducing Styles

Think of styles as themes in your mobile phone. They change the whole look and feel of the GUI application. We're bringing the capability to add VCL and FMX styles to Python with **`delphivcl`** and **`delphifmx`** package distributions respectively.

VCL Styles

Creating a GUI application with DelphiVCL for Python uses the default "**Windows**" VCL style. We can load any other style using the **`StyleManager`** class. The VCL style files end with a **`.vsf`** extension. By adding just five more lines of code, we give our DelphiVCL version of the TODO application a stunning look.

Note: You need a VCL style file **`<style-file>.vsf`** to follow the below demonstration. The eBook bundle includes some styles. Additional styles are available with the Delphi IDE, and you can find other styles online, such as at delphistyles.com.

Now let's add the below two lines at the start of the TODO app's Python code:

```
import os

FILE_DIR = os.path.dirname(os.path.abspath(__file__))
```

We import the **`os`** module to get the current Python file folder path and to load the VCL style file into the GUI application. The global variable **`FILE_DIR`** stores the path of the directory of the TODO app's Python file.

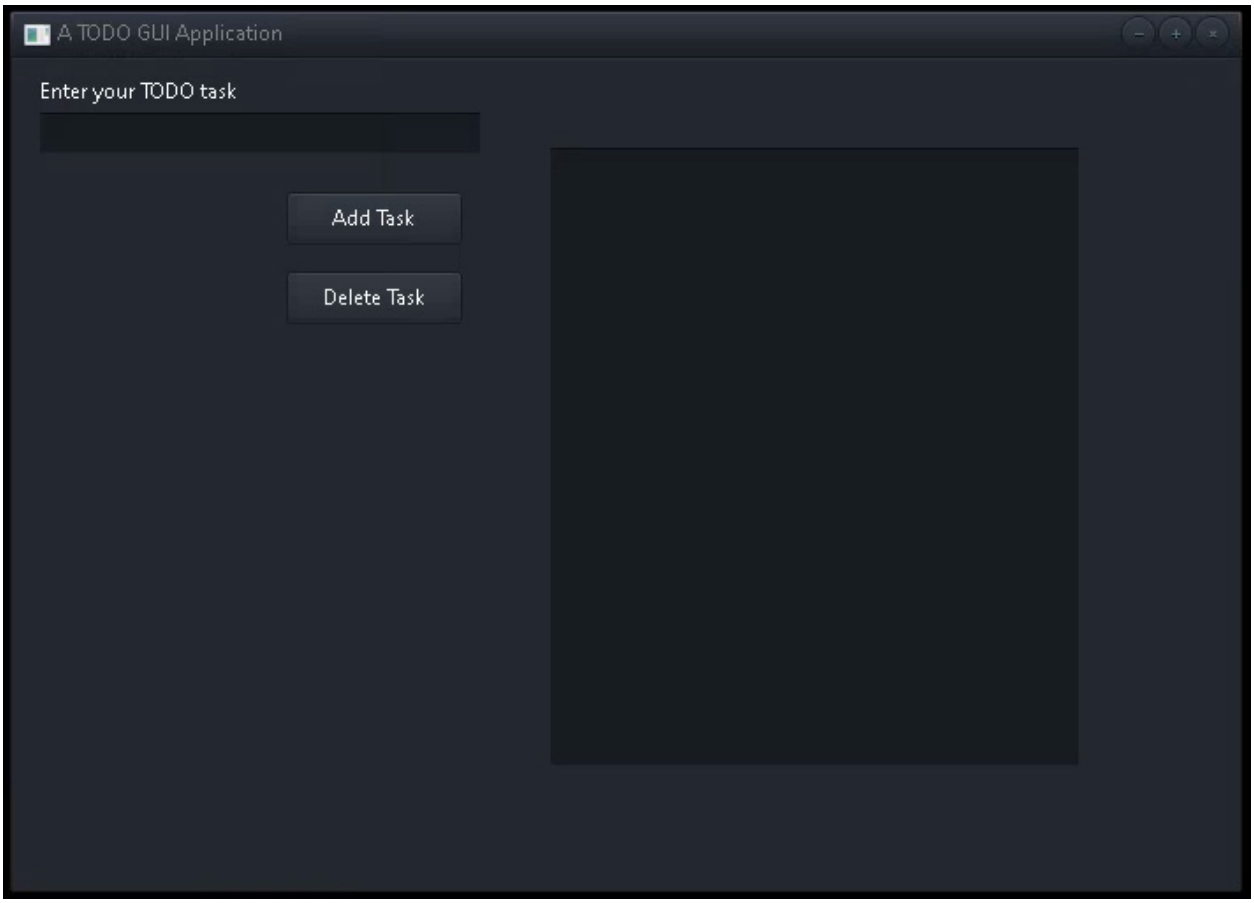
Then add the following three lines of code anywhere inside the **`__init__()`** method's scope of the **`TodoApp`** class.

```
        self.__sm = StyleManager()
        self.__sm.LoadFromFile(os.path.join(FILE_DIR, "Glow.vsf"))
        self.__sm.SetStyle(self.__sm.StyleNames[1])
```

In the above code, we're instantiating **`StyleManager`** in the first line. The second line loads a **`Glow.vsf`** style file that exists in the TODO app's Python file's directory. As mentioned, you need a VCL style file to implement the second line in the above code. Finally, the third line of the above code sets the loaded style for the GUI application.

The **`self.__sm.StyleNames`** stores the list of styles that are available for the GUI application. We already have a default **`Windows`** style at **`0`** index. So the newly loaded style attaches to it at **`1`** index.

As you update the mentioned additions to the TODO app's Python code and run it, you'll see the GUI application's different look and feel. Here, it changes as per the **`Glow.vsf`** style as follows:

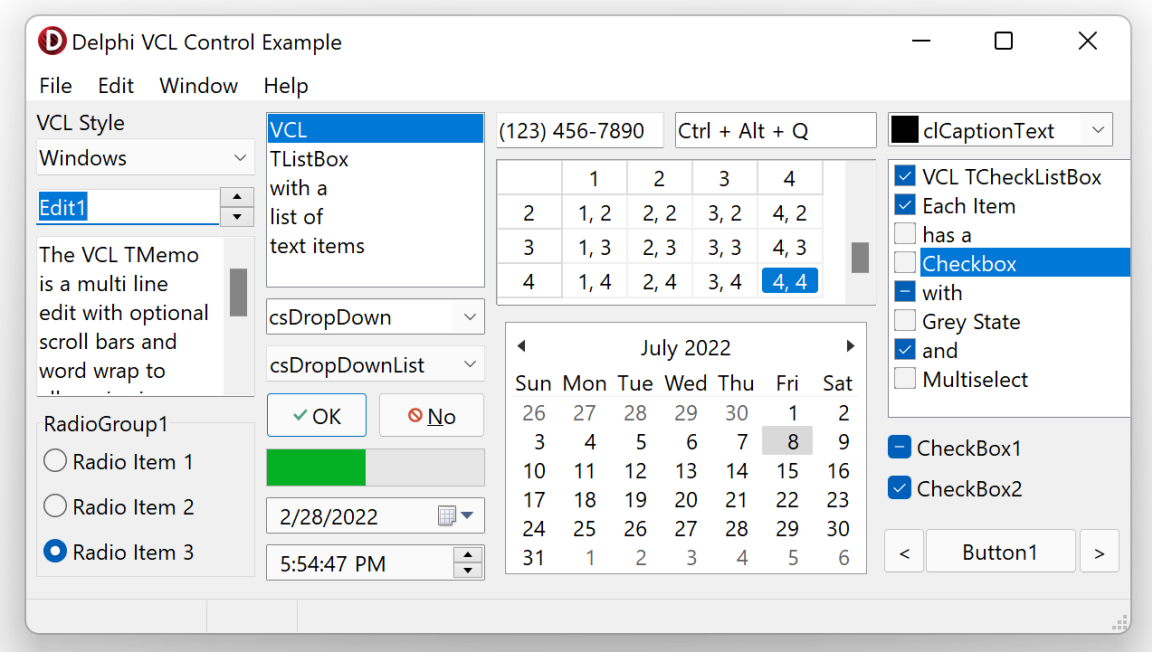


List of included VCL Styles

The following VCL styles are included with this eBook guide and may be distributed with the applications you build with DelphiVCL. The styles all support high-DPI resolutions.

Windows Default

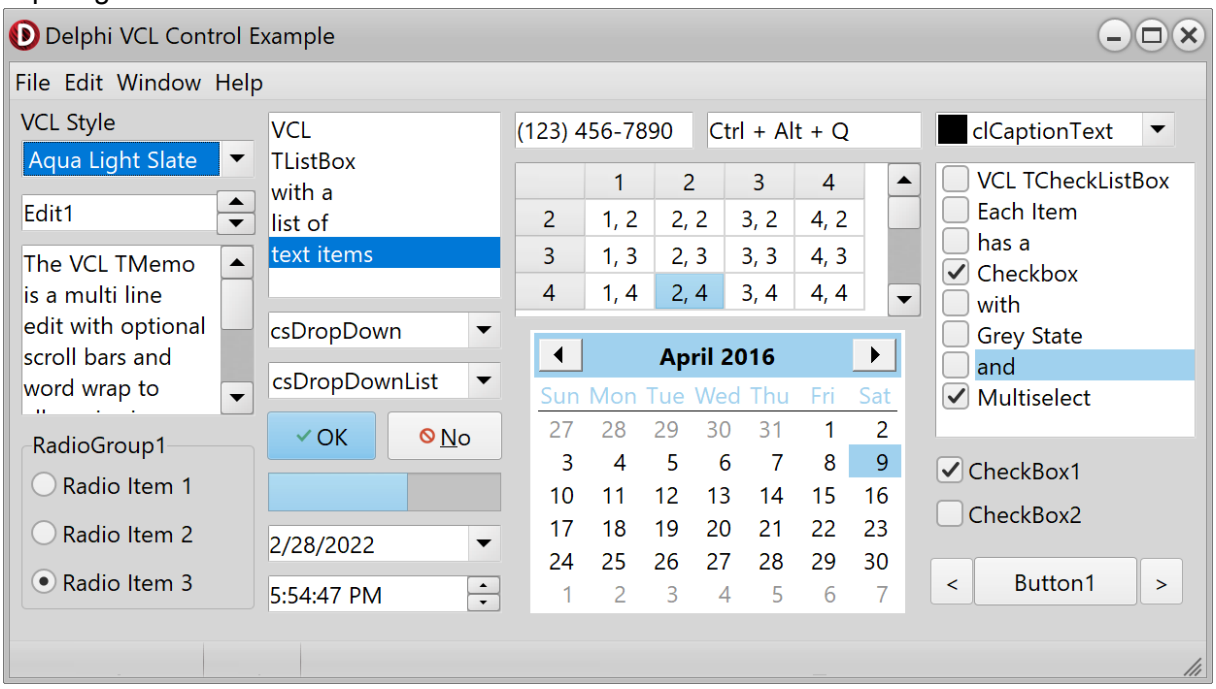
When no style file is included, your DelphiVCL application will use the default style.



This eBook bundle includes the following styles. If you install Delphi, it comes with additional styles and the Bitmap Style Designer that allows you to create and customize styles. You can find out more about Embarcadero Delphi and its different editions at embarcadero.com/products/delphi

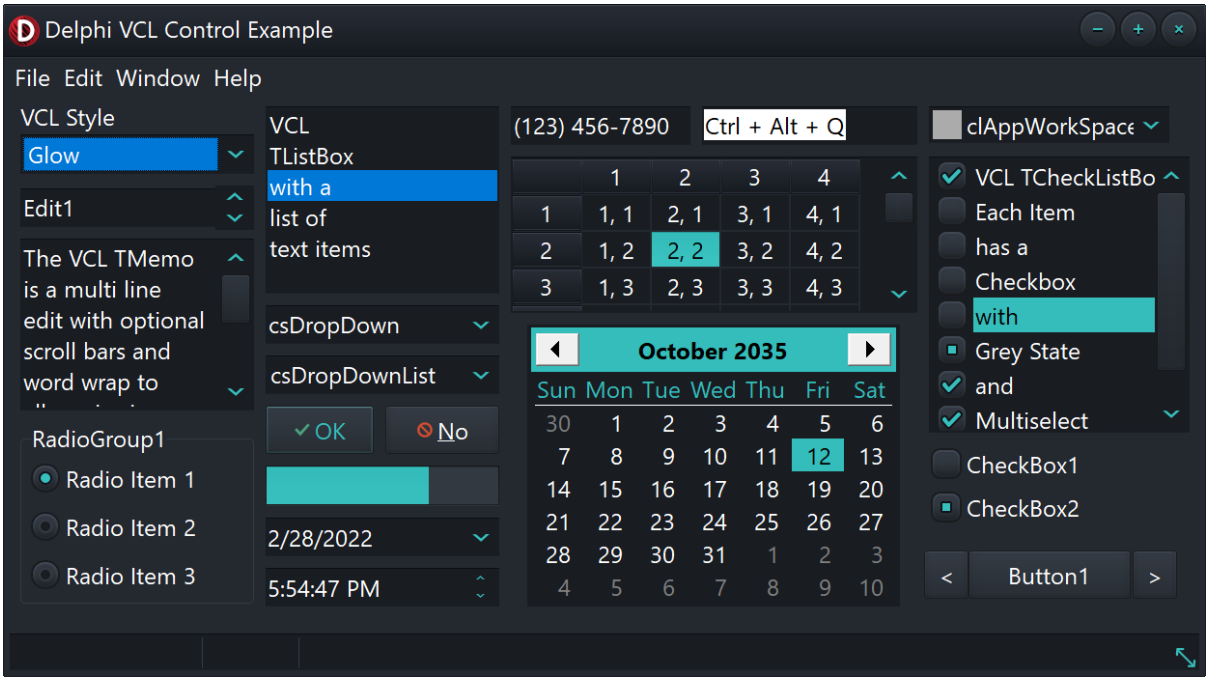
Aqua Light Slate

AquaLightSlate.vsf



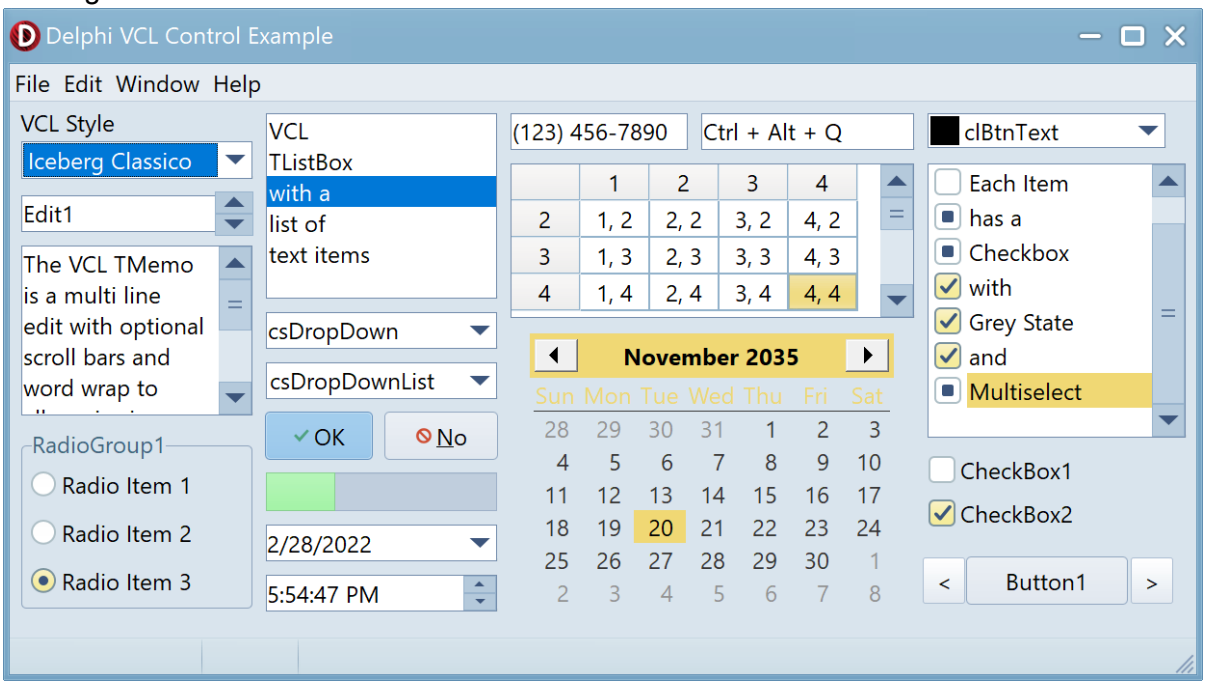
Glow

Glow.vsf



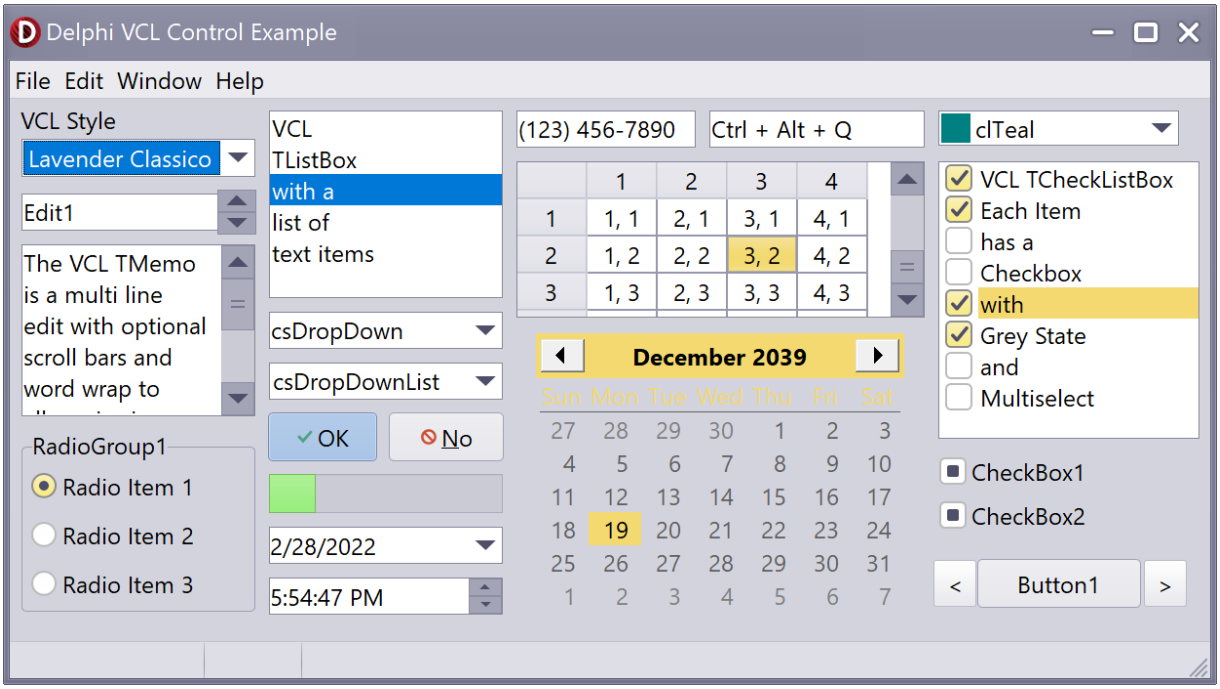
Iceberg Classico

IcebergClassico.vsf



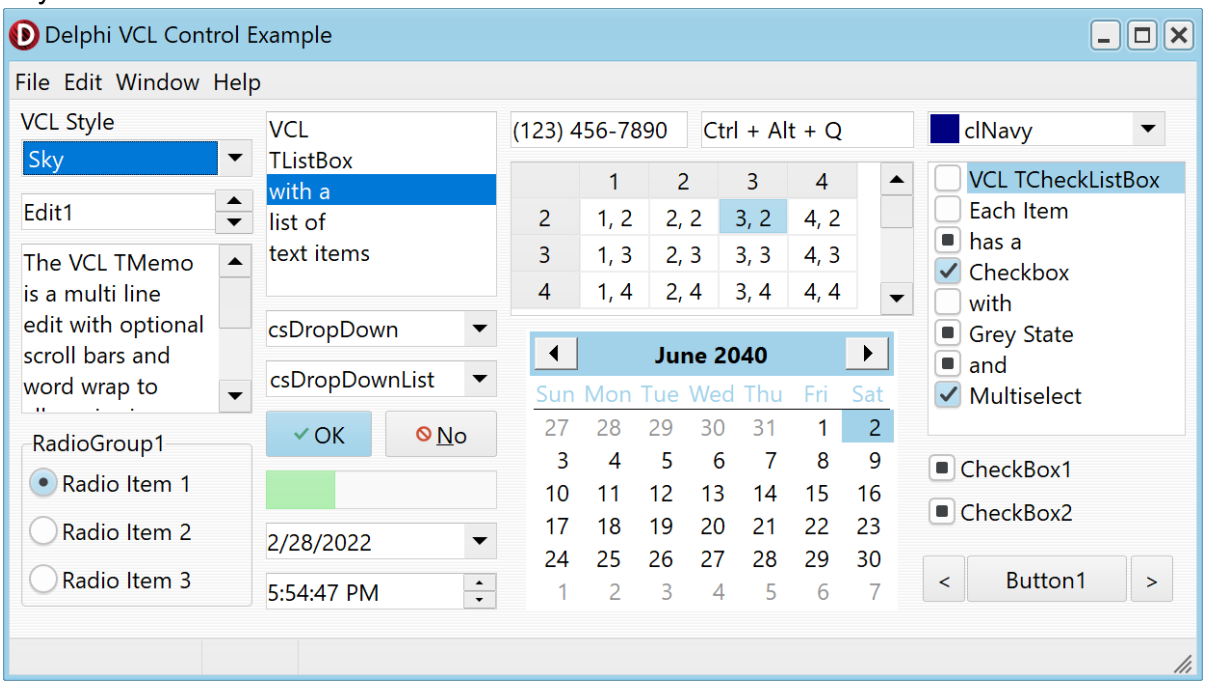
Lavender Classico

LavenderClassico.vsf



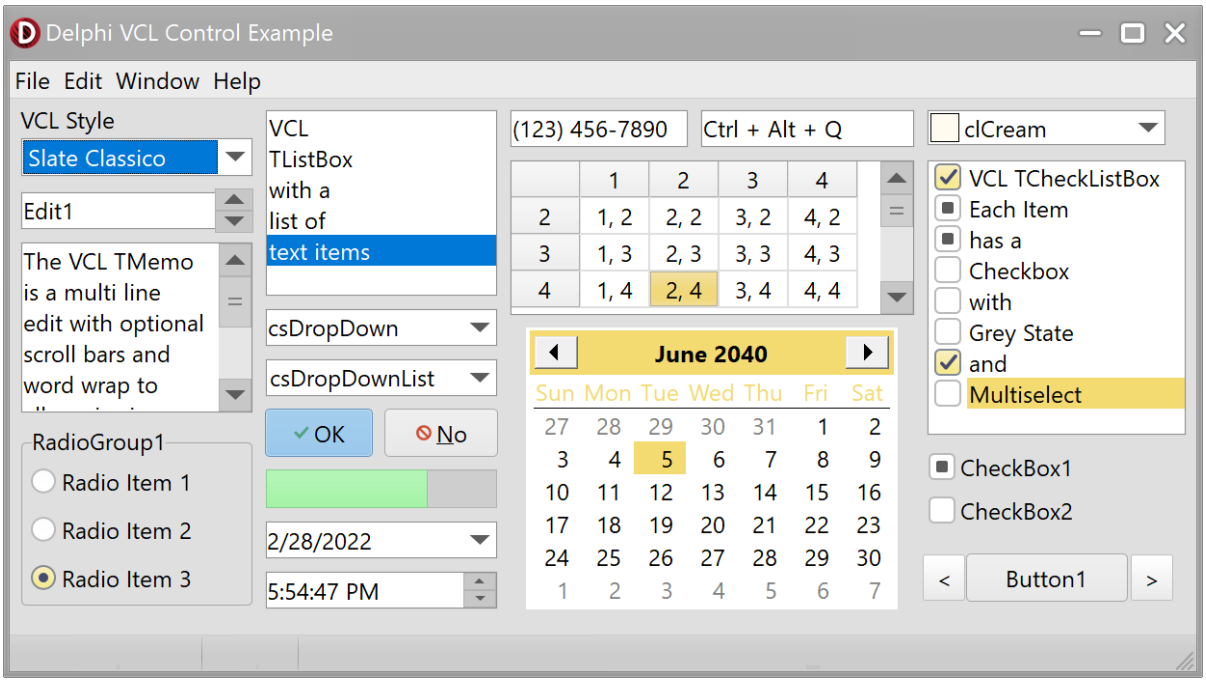
Sky

Sky.vsf



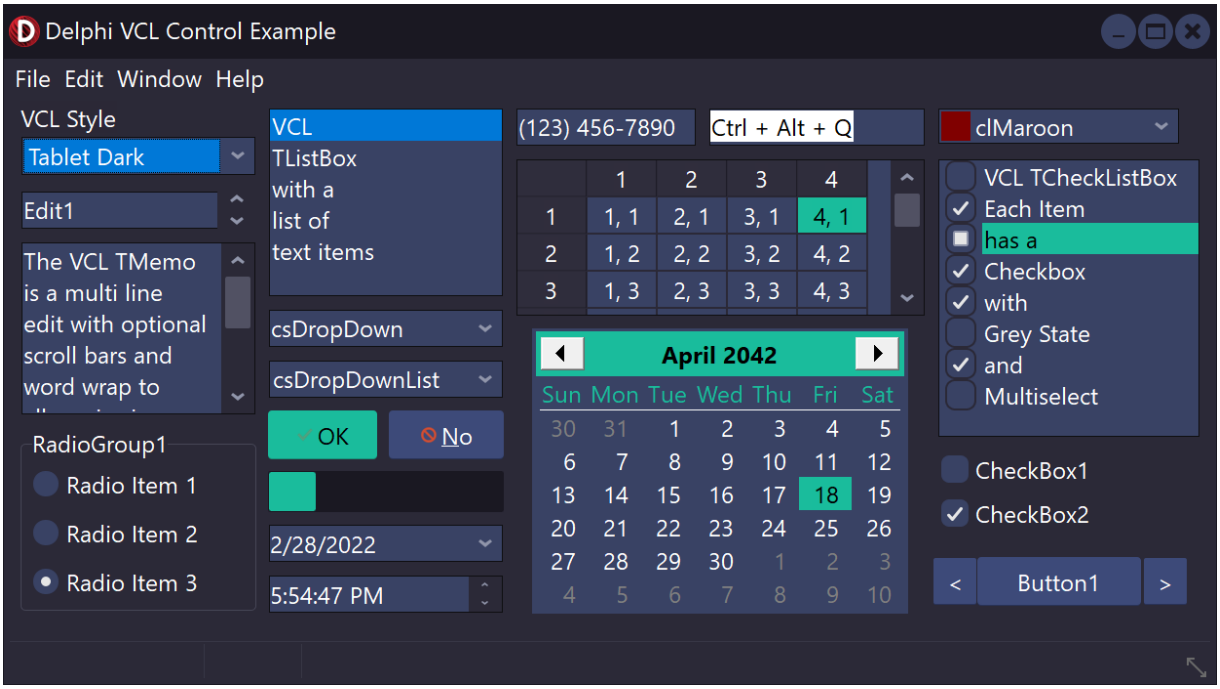
Slate Classico

SlateClassico.vsf



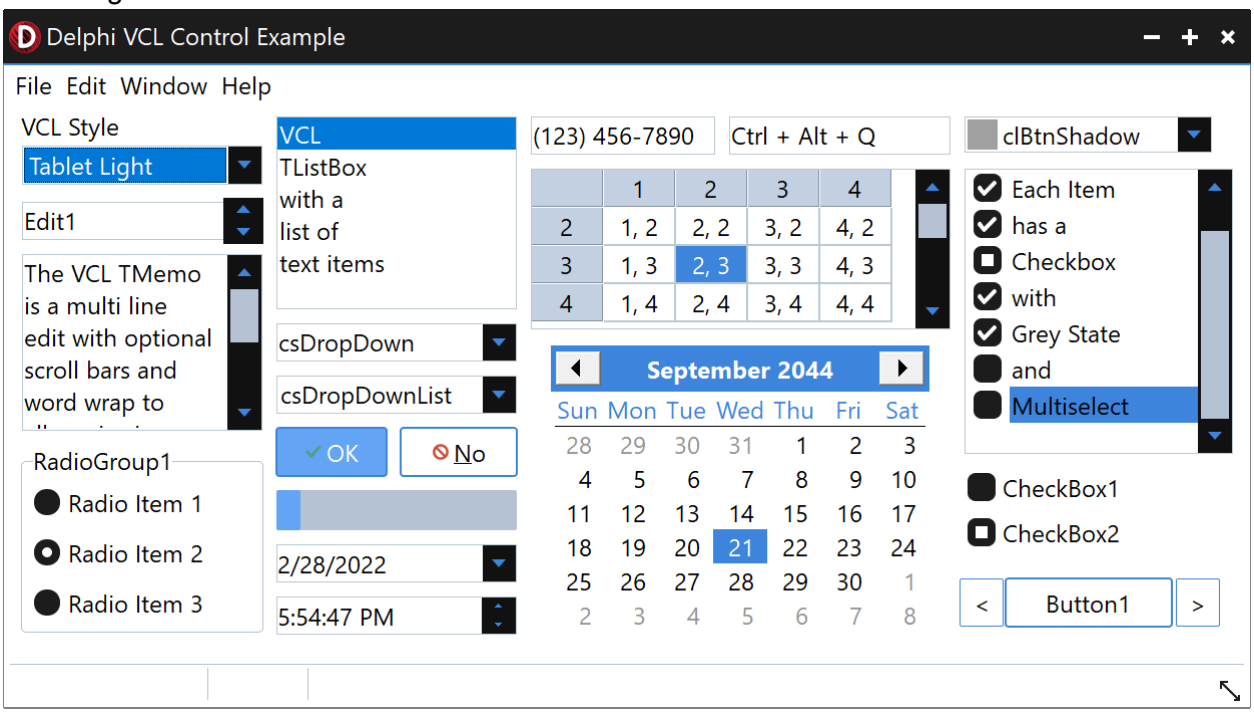
Tablet Dark

TabletDark.vsf



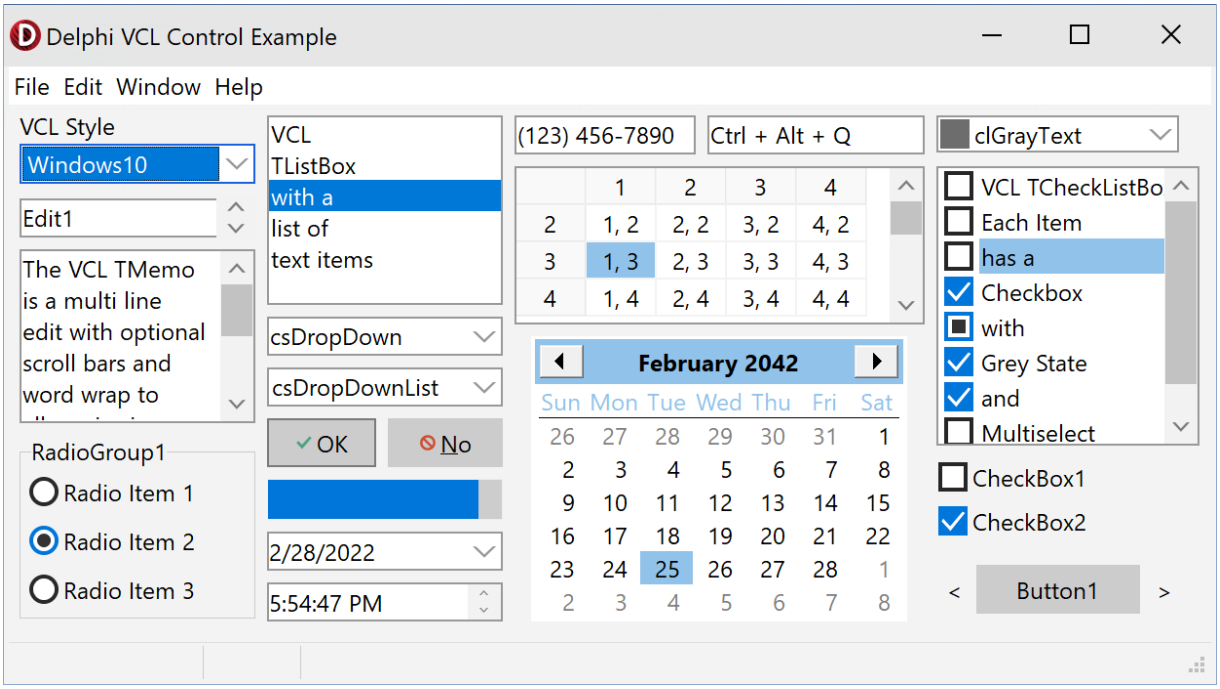
Tablet Light

TabletLight.vsf



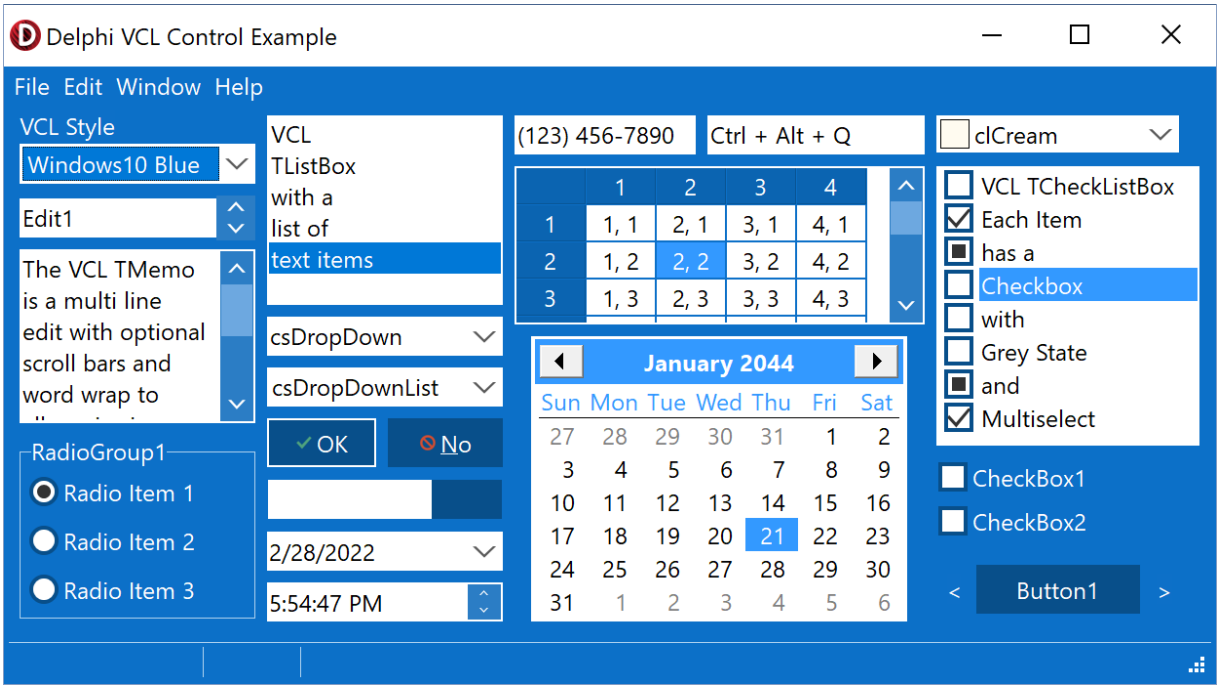
Windows 10

Windows10.vsf



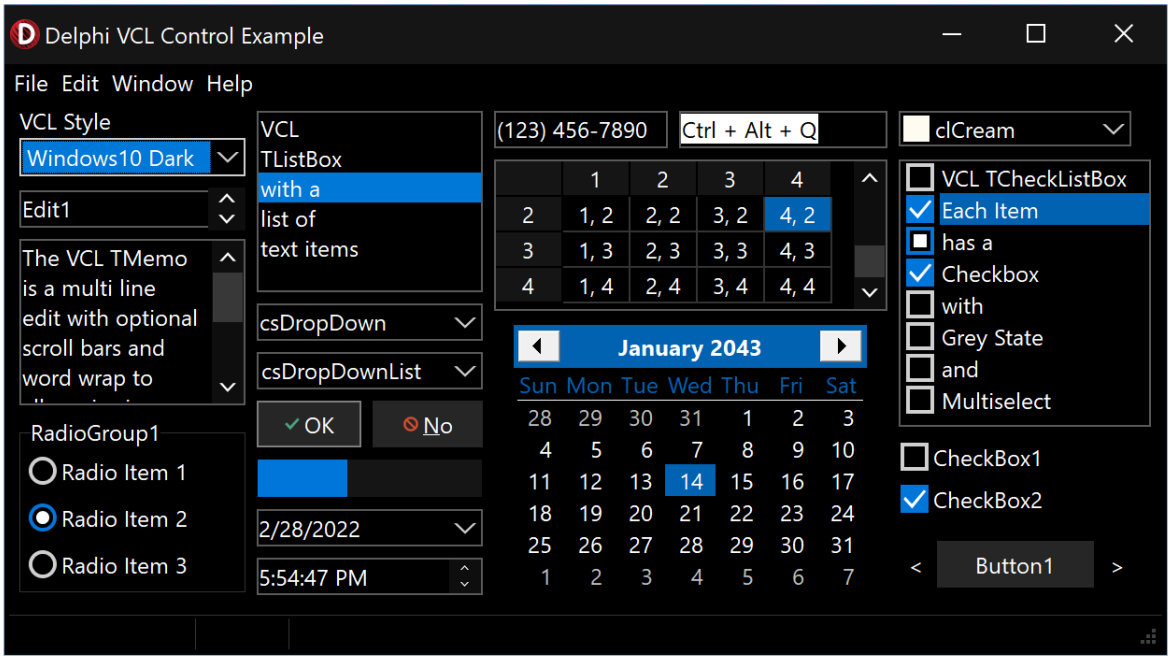
Windows 10 (Blue)

Windows10Blue.vsf



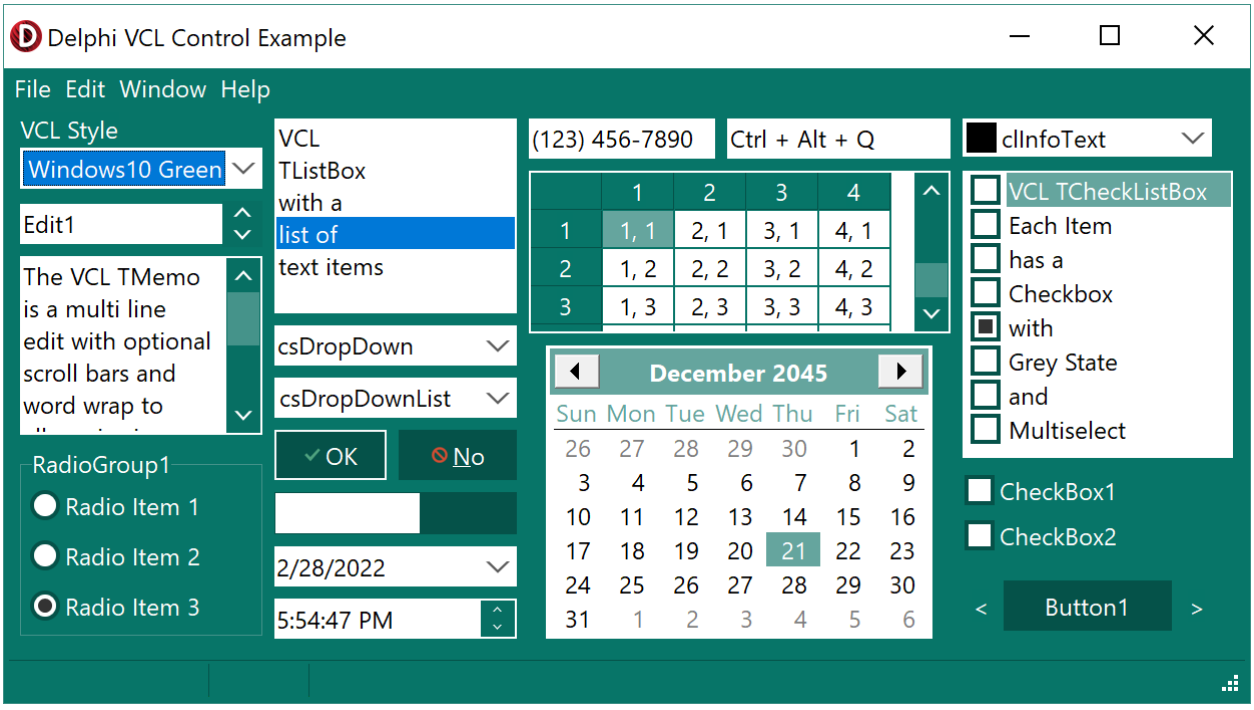
Windows 10 (Dark)

Windows10Dark.vsf



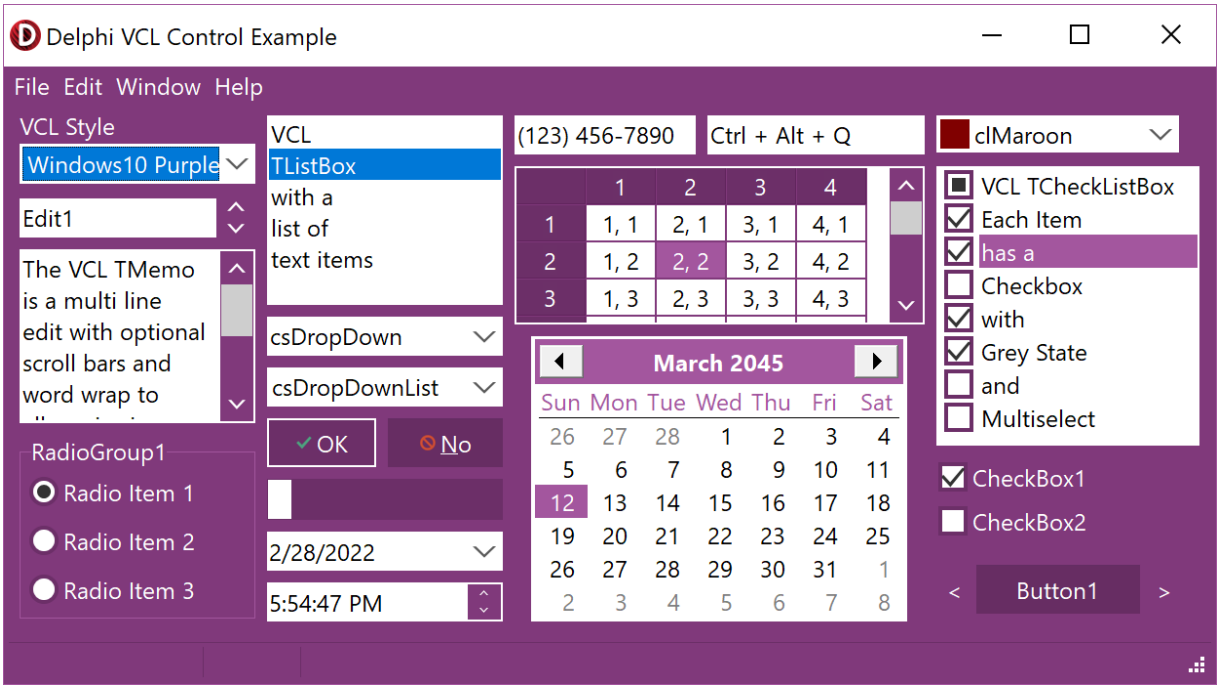
Windows 10 (Green)

Windows10Green.vsf



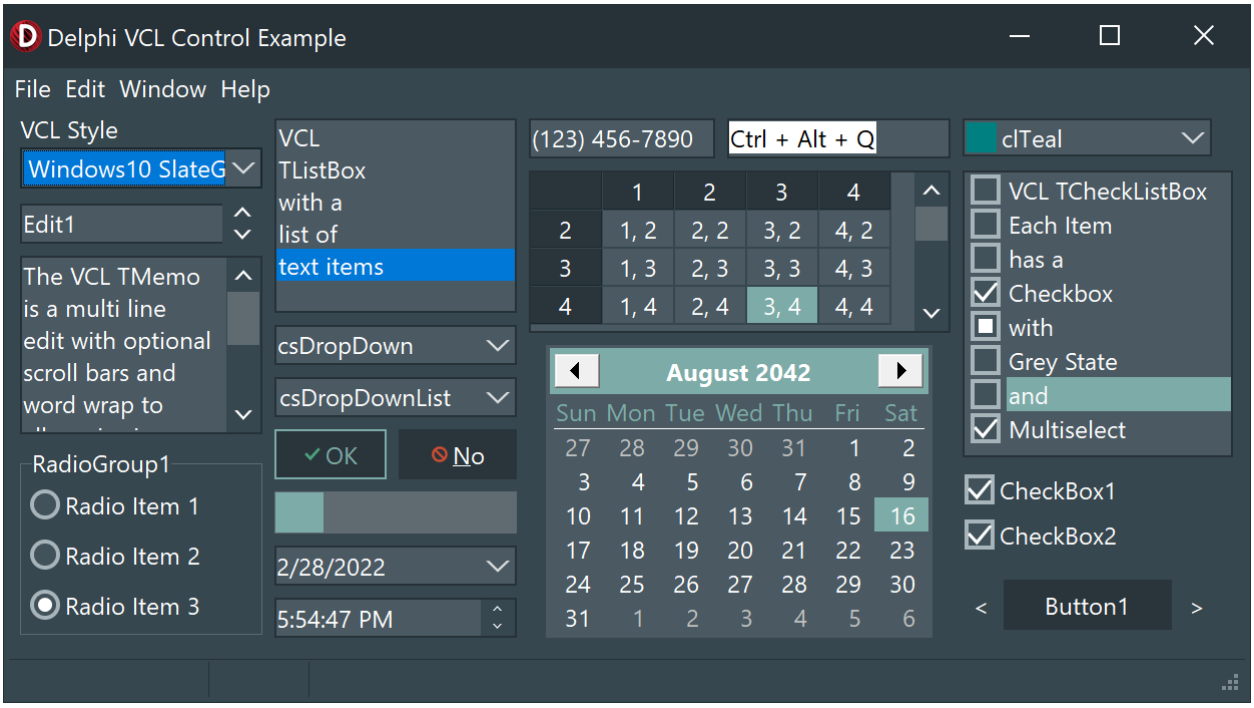
Windows 10 (Purple)

Windows10Purple.vsf



Windows 10 (Slate Gray)

Windows10SlateGray.vsf



FMX Styles

Creating a GUI application with DelphiFMX for Python uses the default FMX style for the operating system that you're working on. We can load any other style using the **`StyleManager`** and **`StyleStreaming`** classes. The FMX style files end with the **`.style`** extensions. By just adding three more lines of code to the FMX version of TODO application, we give it a stunning look.

Note: You need an FMX style file **`<style-file>.style`** to follow the below demonstration. The eBook bundle includes some styles. Additional styles are available with the Delphi IDE, and you can find other styles online, such as at delphistyles.com.

Now let's add the below two lines at the start of the DelphiFMX version of TODO app's Python code:

```
import os
```

We import the **`os`** module to get the current Python file folder path and to load the FMX style file into the GUI application.

Then add the following three lines of code anywhere inside the **`__init__()`** method's scope of the **`TodoApp`** class.

```
self.__sm = StyleManager()
self.__sm.SetStyle(StyleStreaming().LoadFromFile(os.path.join(os.getcwd(),
"Transparent.syle")))
```

In the above code, we're instantiating **`StyleManager`** in the first line. The second line loads a **`Transparent.Style`** style file that exists in the TODO app's Python file's directory using the **`StyleStreaming`** class. As mentioned, you need an FMX style file to implement the second line in the above code. Finally, the **`SetStyle()`** method in the above code sets the loaded style for the GUI application.

As you update the mentioned additions to the TODO app's Python code and run it, you'll see the GUI application's different look and feel. Here, it changes as per the **`Transparent.Style`** style as follows:

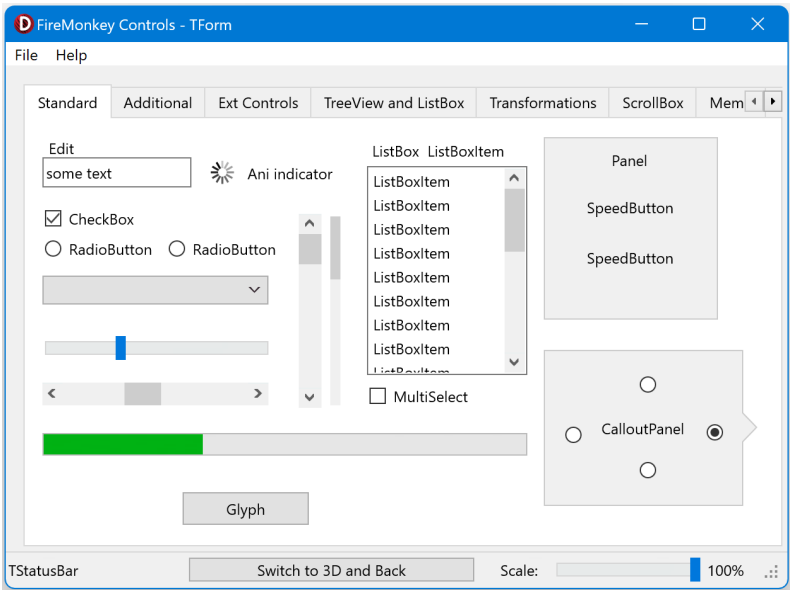

A TODO GUI Application
Enter your TODO task
Add Task
Delete Task

List of included FMX styles

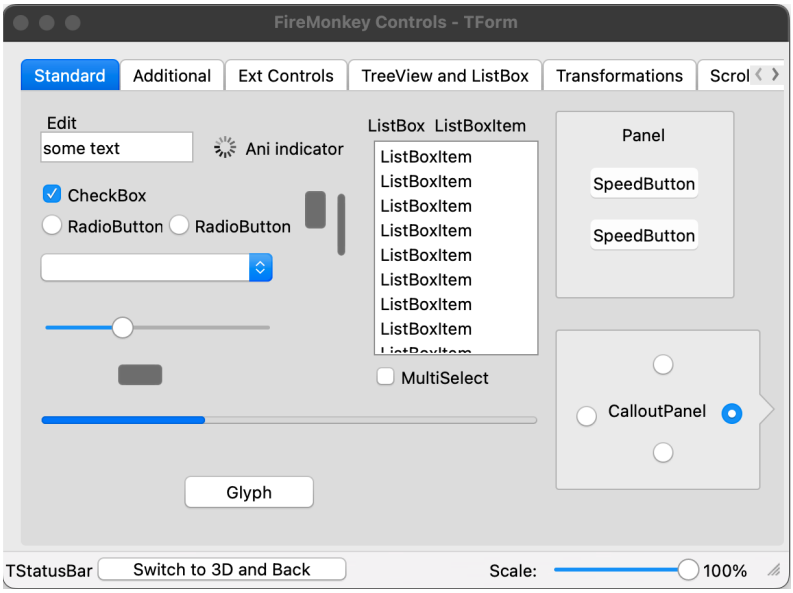
The following FMX styles are included with this eBook and may be distributed with the applications you build with DelphiFMX.

Default

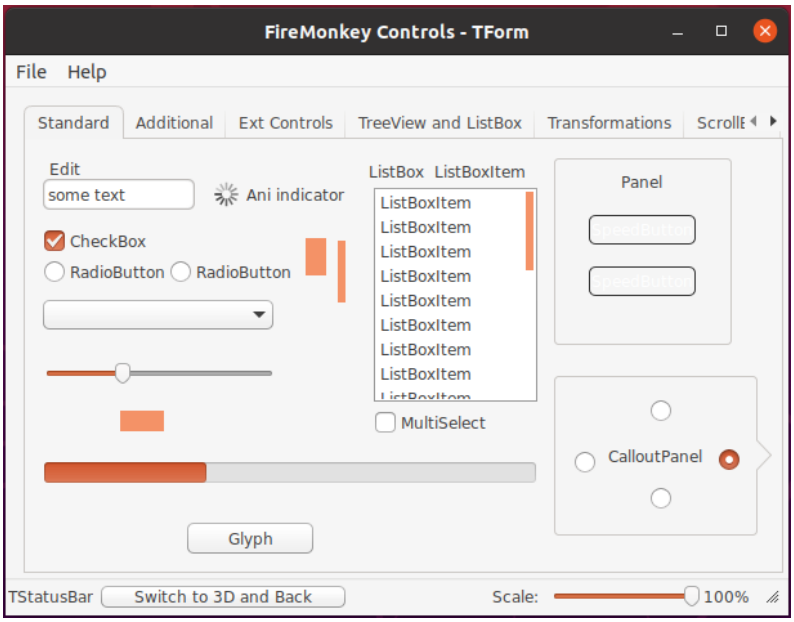
When no style file is included, your DelphiFMX application will use the default style. Here is what the default style looks like on Windows.



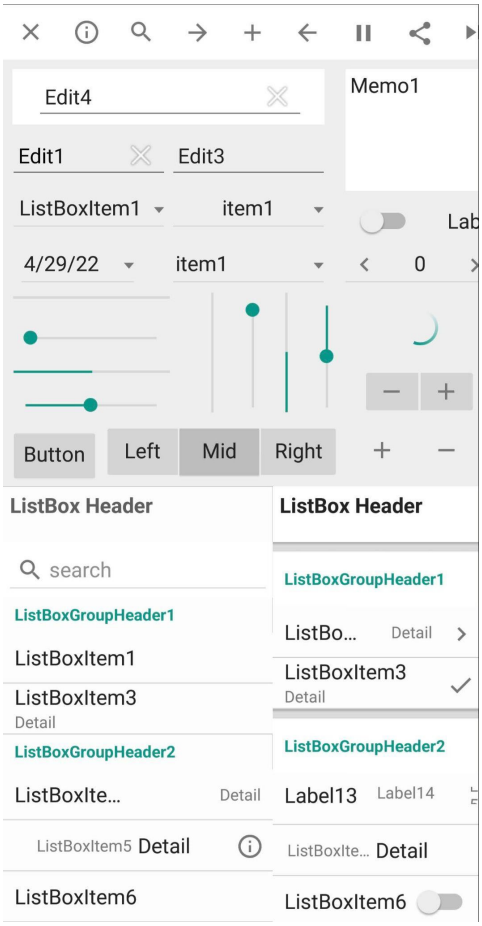
Here is the default style on macOS:



And here it is on Ubuntu Linux:



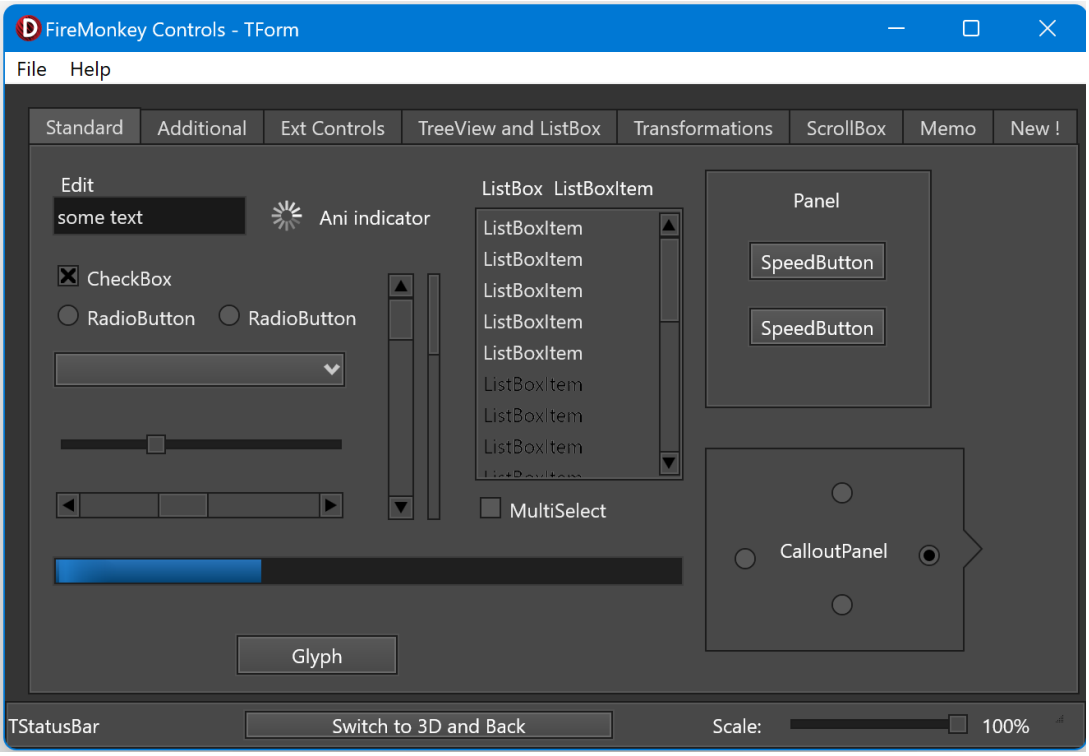
And here it is on Android:



This eBook bundle includes the following styles. We are including screenshots from Windows to give you a general idea, but they may look slightly different on other platforms. If you install Delphi (even the free Community Edition), it comes with additional styles and allows you to create and customize styles. You can find out more about Embarcadero Delphi and its different editions at embarcadero.com/products/delphi

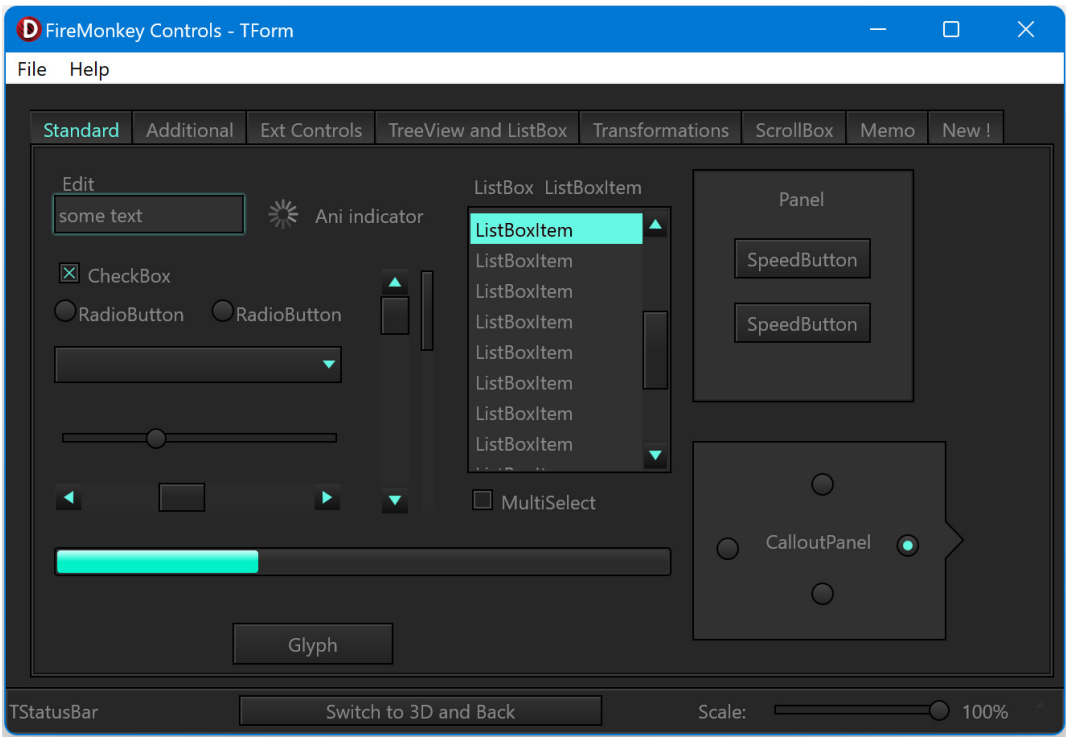
Air

Air.style



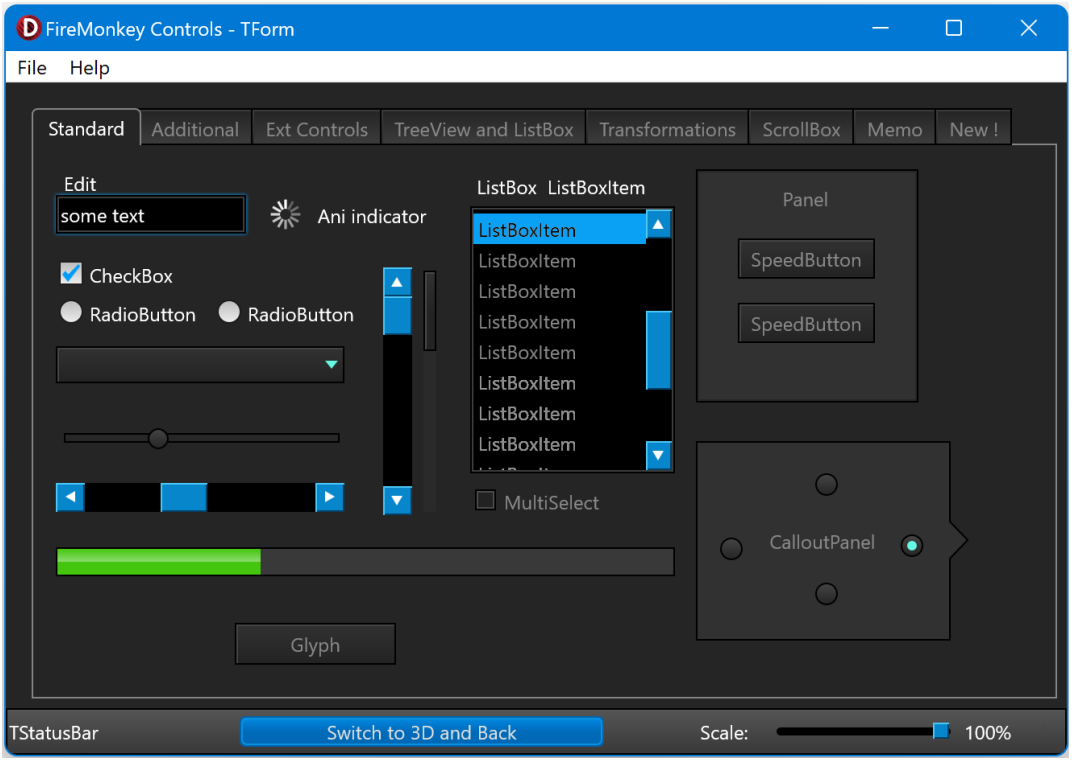
Amakrits

Amakrits.style



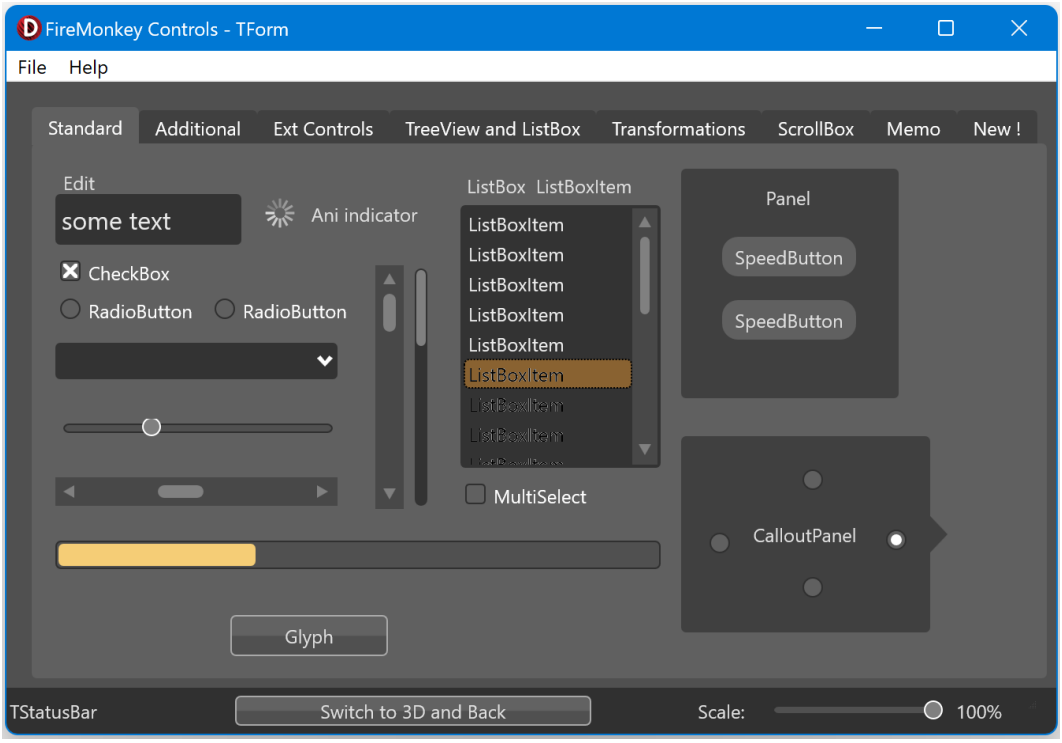
Aqua Graphite

AquaGraphite.style



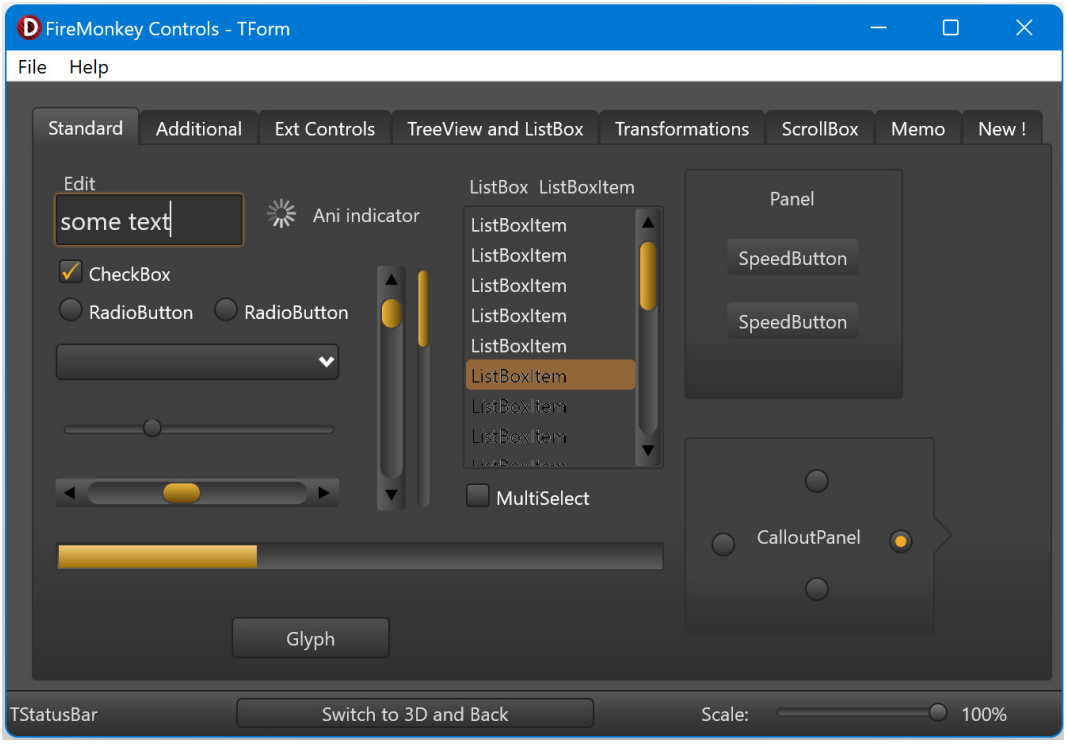
Blend

Blend.style



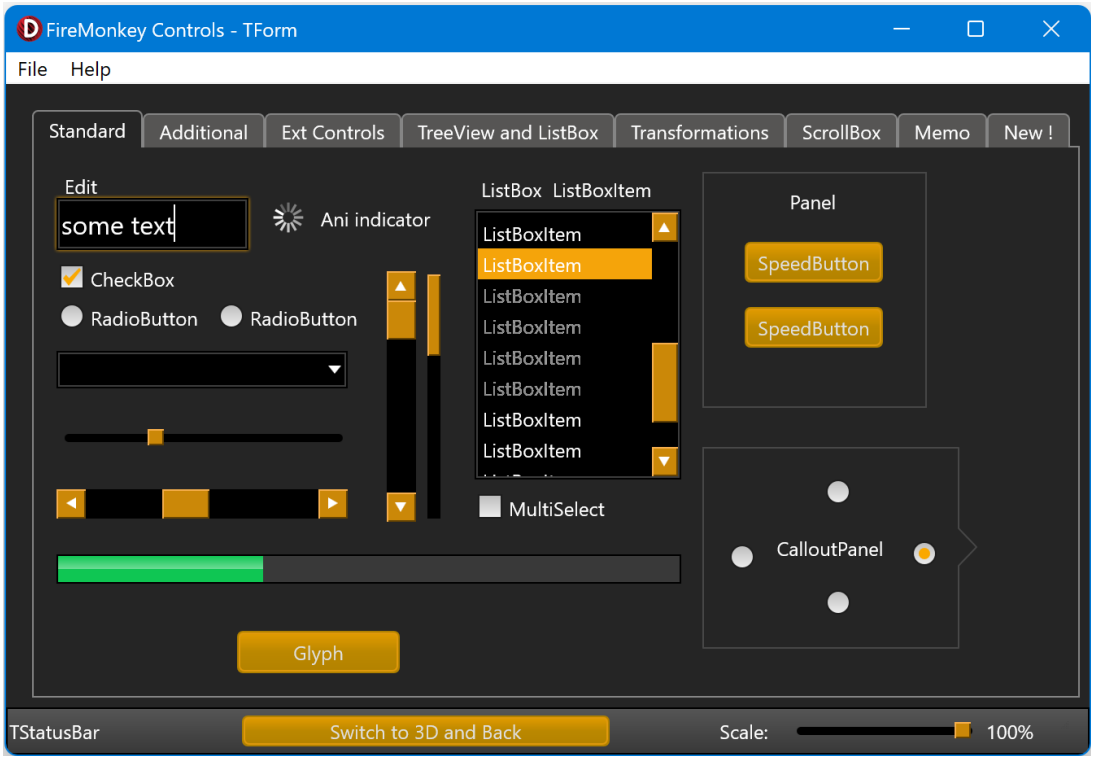
Dark

Dark.style



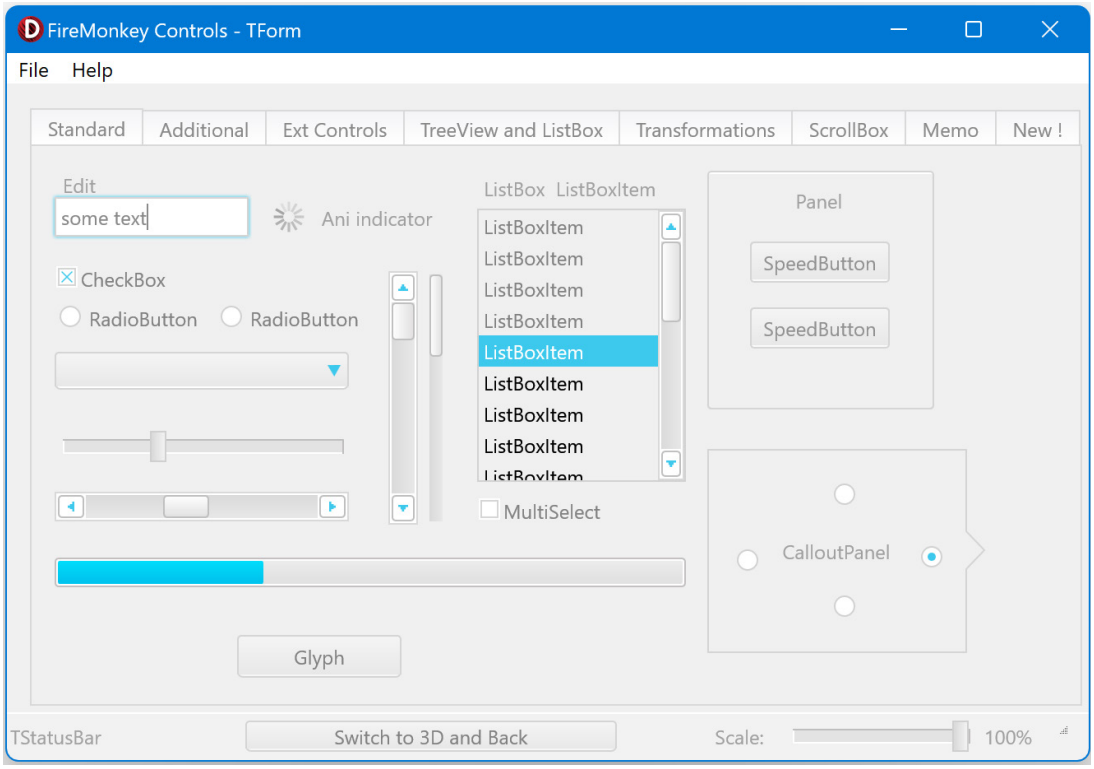
Golden Graphite

GoldenGraphite.style



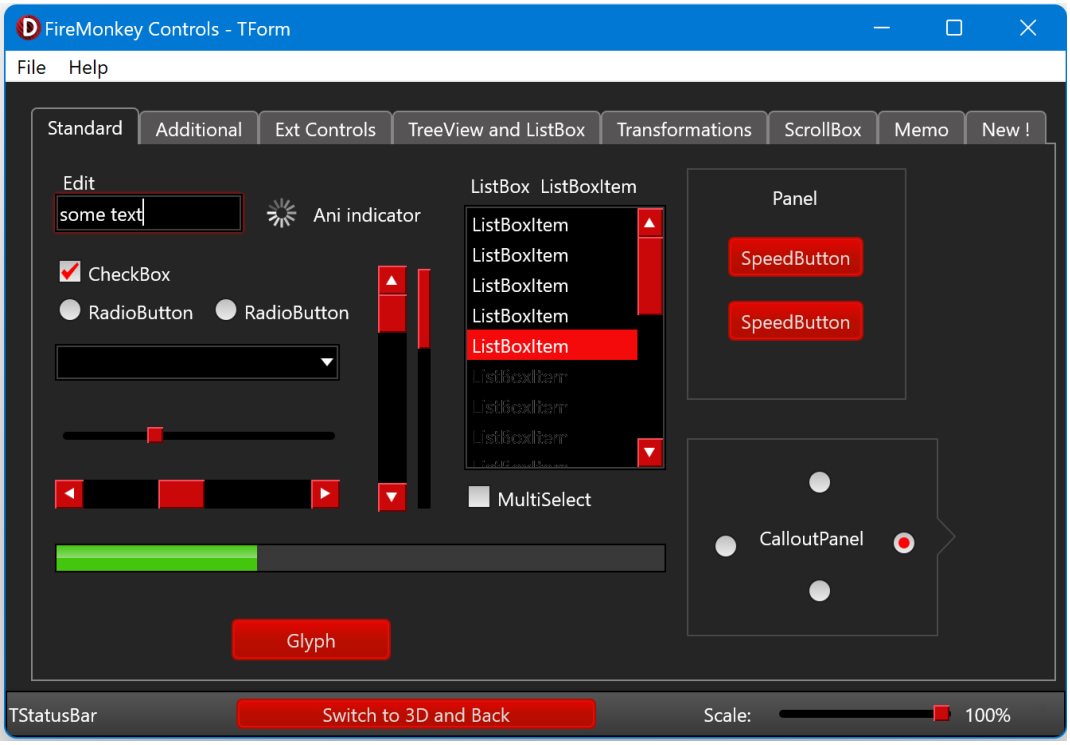
Light

Light.style



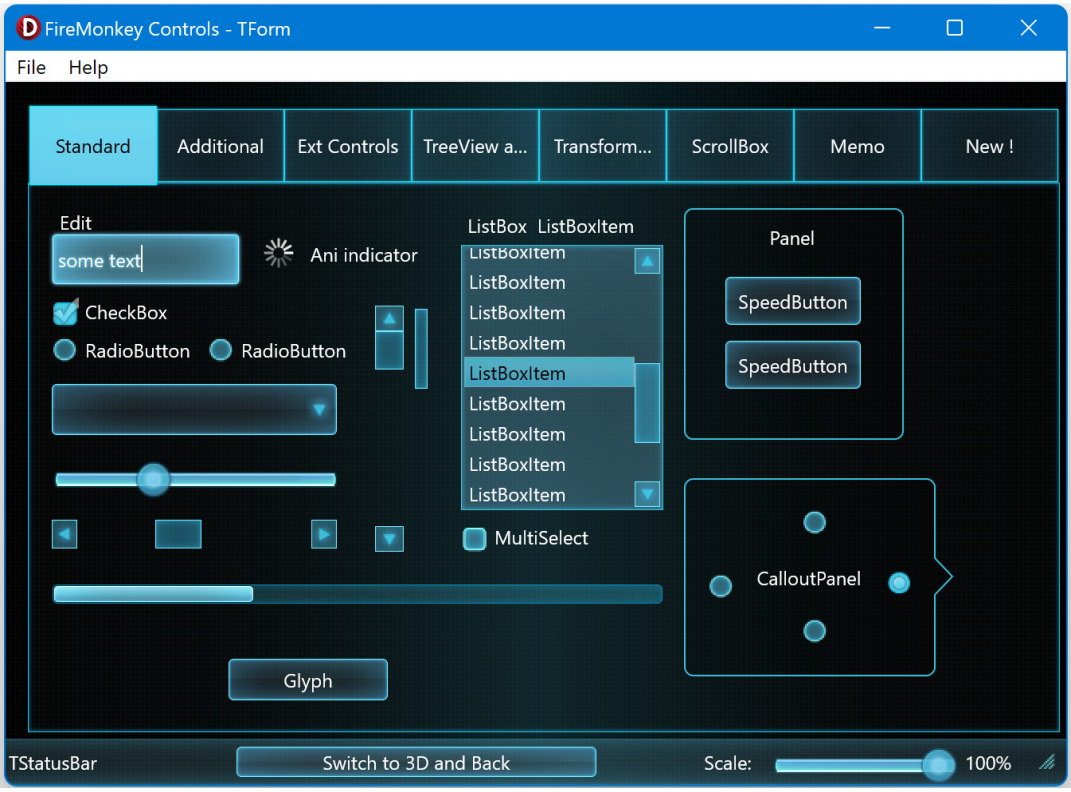
Ruby Graphite

RubyGraphite.style



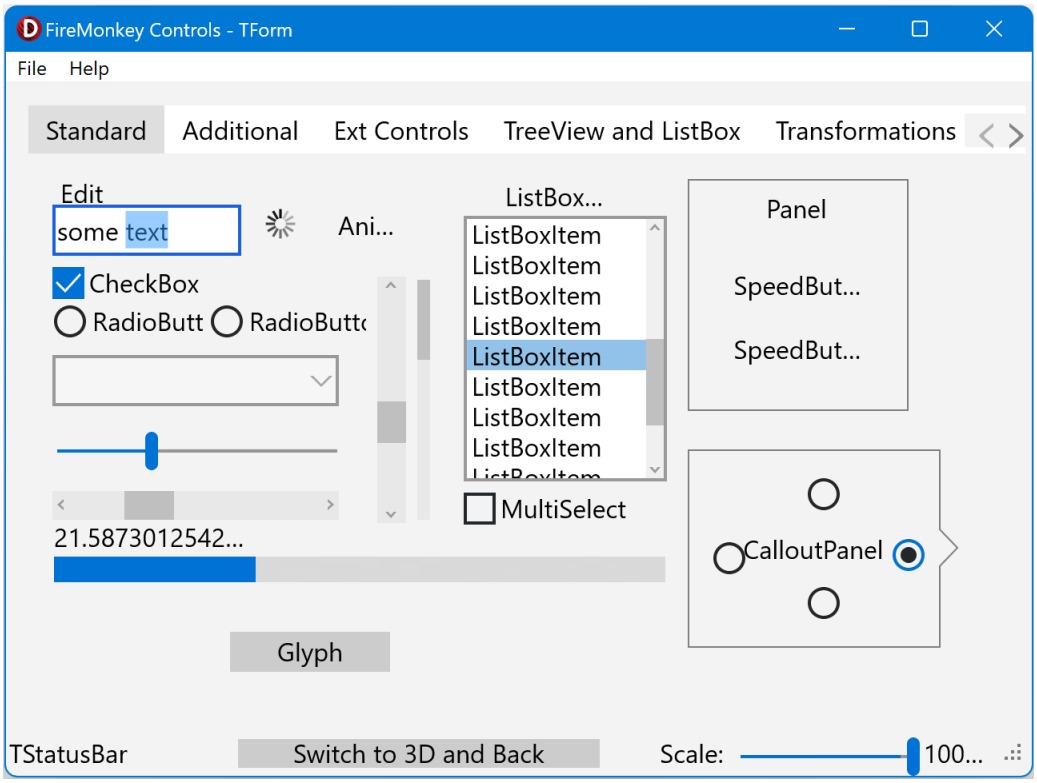
Transparent

Transparent.style



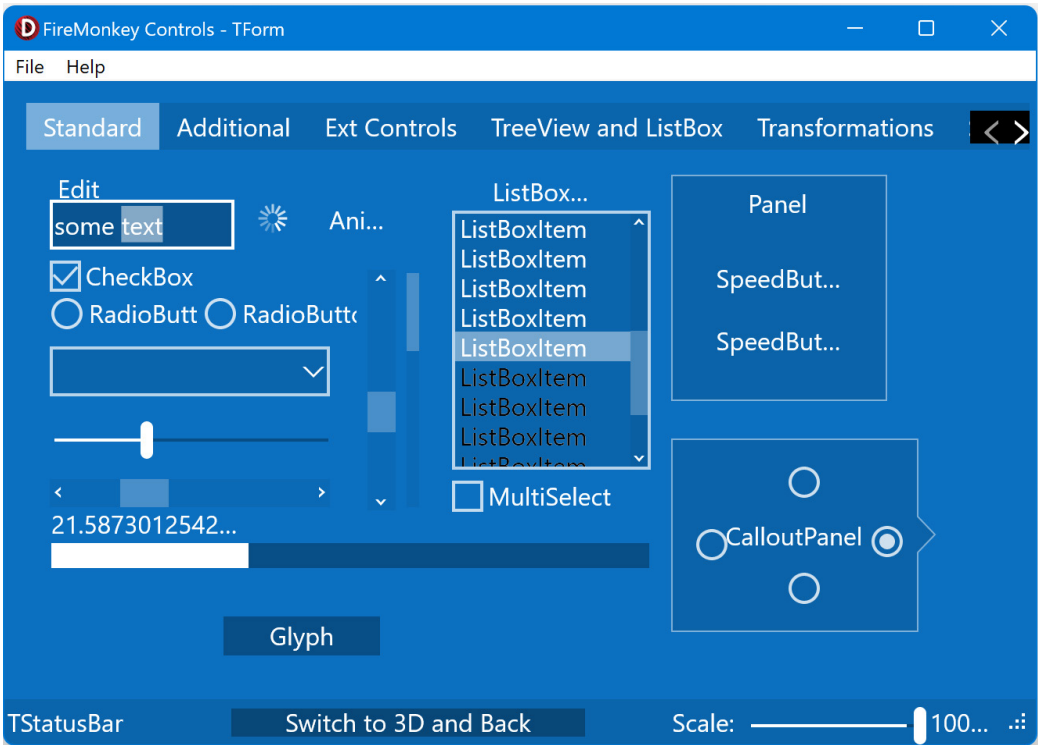
Windows Modern

WindowsModern.style



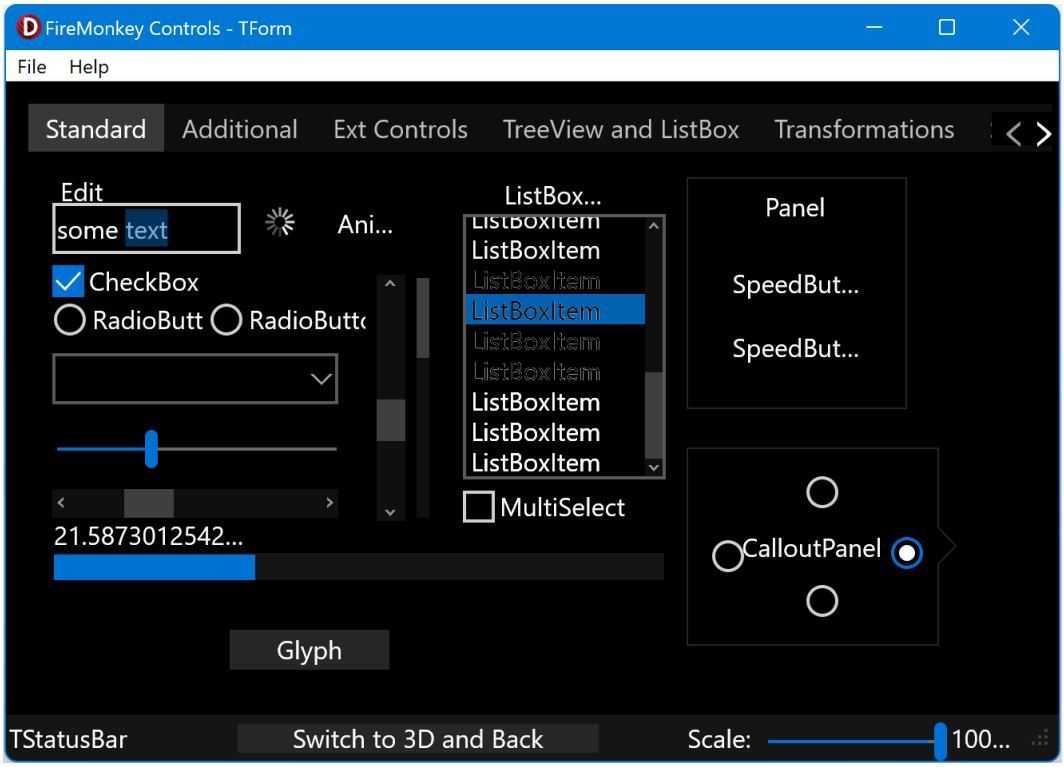
Windows Modern Blue

WindowsModernBlue.style



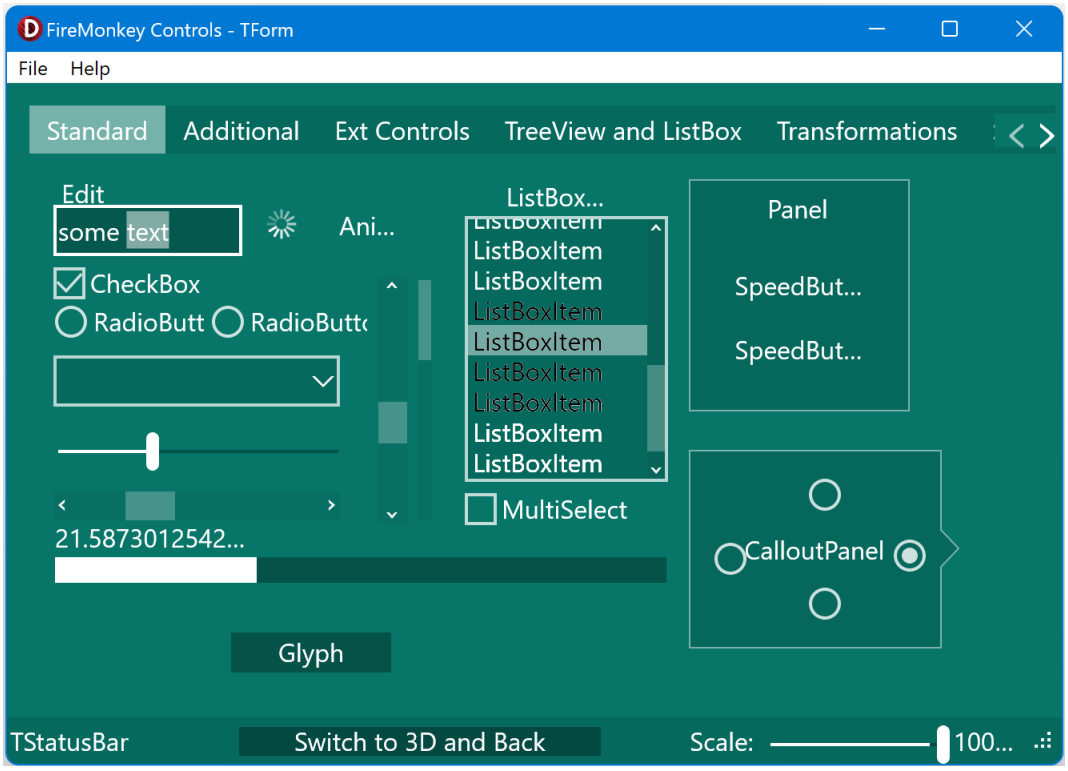
Windows Modern Dark

WindowsModernDark.style



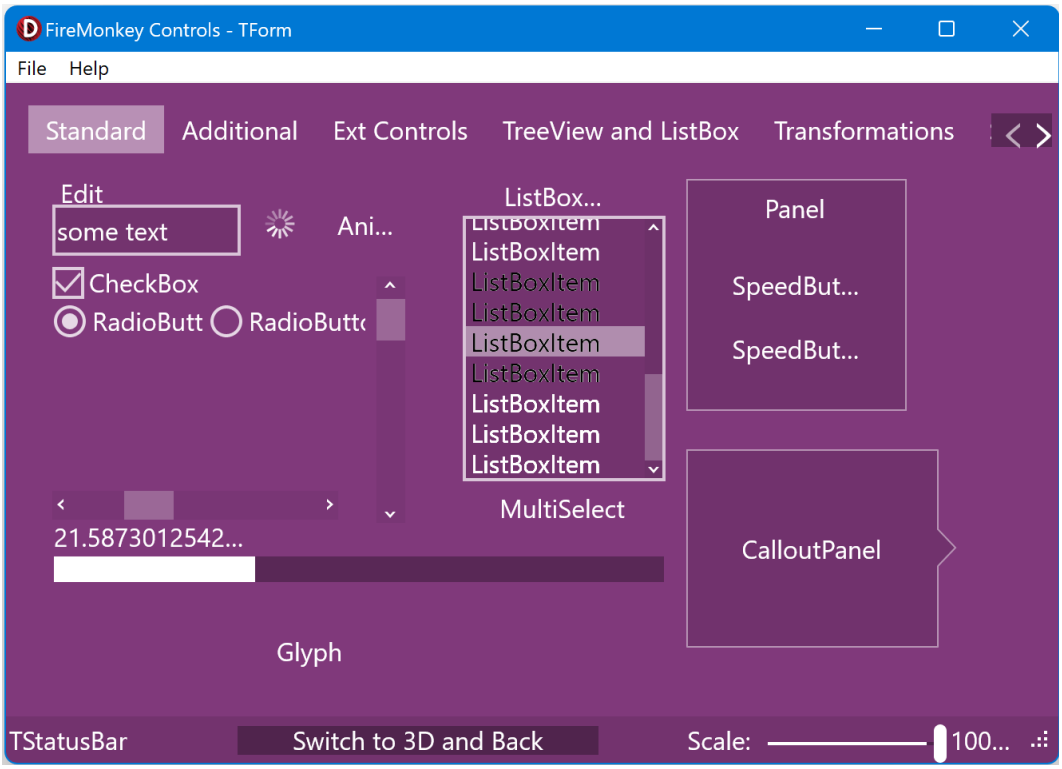
Windows Modern Green

WindowsModernGreen.style



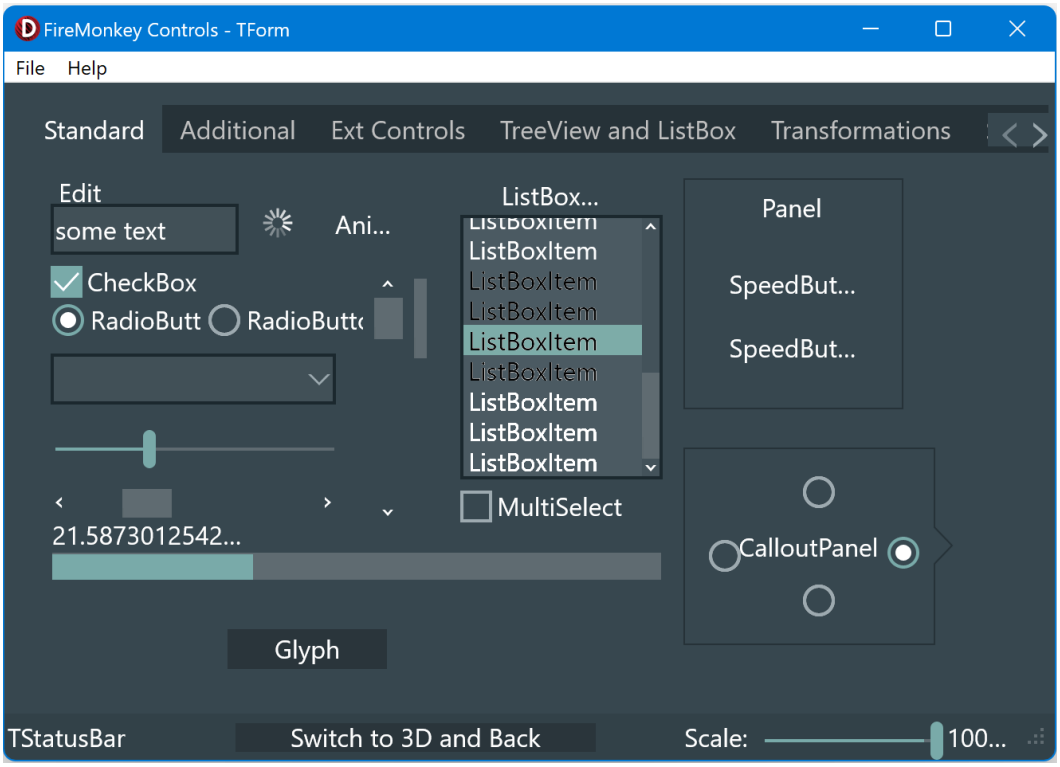
Windows Modern Purple

WindowsModernPurple.style



Windows Modern SlateGray

WindowsModernSlateGray.style



Summary

Python bindings for Delphi GUI libraries are essential for the Python GUI developer community. We're bringing the research and development of the mature Delphi FMX GUI library to Python. Some of the professional-grade GUI applications built using Delphi include:

- FL Studio/Fruity Loops Digital Audio Workstation: image-line.com/flstudio/
- KMPlayer Media player: kmplayer.com
- PyScripter: pyscripter.dev

For more case studies, please check
Embarcadero success stories
www.embarcadero.com/case-study

About Embarcadero Technologies

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster, and run them better regardless of platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance, and accelerate innovation. Founded in 1993, Embarcadero is headquartered in Austin, Texas, with offices located around the world.

10801 North Mopac Expressway, Building 1, Suite 100, Austin, TX, 78759
www.embarcadero.com/company/contact-us
US: 1 (512) 226-8080 - info@embarcadero.com

About PyScripter

PyScripter is an open-source and feature-rich lightweight Python IDE.

PyScripter has all the features expected in a modern Python IDE in a lightweight package. It's also natively compiled for Windows to use minimal memory with maximum performance. The IDE is open-source and fully developed in Delphi with extensibility via Python scripts.
www.embarcadero.com/free-tools/pyscripter/free-download

About Delphi

Delphi is Embarcadero's flagship development tool supporting native application and server development for Windows, macOS, Linux, Android, and iOS. It includes a variety of libraries, including a robust framework for database applications, REST services, and visual application development. It is available in multiple editions, including a free Community Edition, an Academic Edition, and Professional, Enterprise, and Architect Editions.
www.embarcadero.com/products/delphi

Download a free trial today!