Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Vítězslav Imrýšek

WPF Style GUI Library for MonoGame Framework

Department of Distributed and Dependable Systems

Bachelor thesis supervisor: Mgr. Pavel Ježek, Ph.D.

Study program: Computer Science

Specialization: General Computer Science

Prague 2015

Special thanks to Mgr. Pavel Ježek Ph.D. for supervising this thesis, and most of all for his patience and all his invaluable advices.

Název práce: MonoGame knihovna pro tvorbu GUI ve stylu WPF

Autor: Vítězslav Imrýšek

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.,
Katedra distribuovaných a spolehlivých systémů

Abstrakt: MonoGame je populární multi-platformní open-source framework používaný pro vývoj her a dalších grafických aplikací. Nicméně tento framework samotný neposkytuje žádnou implicitní podporu pro vytváření uživatelských prostředí. A zatímco existuje řada knihoven třetích stran, které se tuto podporu snaží poskytnout, žádná z nich nemá za svůj cíl implementaci nějakého existujícího a hojně využívaného frameworku pro tvorbu uživatelských rozhraní.

Tato práce se zaměřuje na tento nedostatek a poskytuje reimplementaci Windows Presentation Foundation (WPF) frameworku ve formě knihovny pro MonoGame. V rámci této práci jsme vybrali vhodnou podmnožinu vlastností, které budou implementovány a rovněž jsme vyřešili i řadu technických problémů, jako jakým způsobem renderovat naše geometrická primitiva, či jak implementovat podporu pro neobdélníkové ořezávání. V průběhu této práce byl kladen velký důraz na co nejpřesnější dodržování existujícího WPF API a jeho chování.

Klíčová slova: MonoGame, WPF, Knihovna pro tvorbu uživatelských rozhraní

Title: WPF Style GUI Library for MonoGame Framework

Author: Vítězslav Imrýšek

Department: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.,
Department of Distributed
and Dependable Systems

Abstract: MonoGame is a popular cross-platform open-source framework used for developing games and other graphical applications. However, this framework has no out-of-box support for user interface creation. And while there exist many third party UI libraries, none of them has the goal of implementing some existing, widely used user interface framework.

For this thesis, we decided to target this shortcoming and reimplement the Windows Presentation Foundation framework, in form of a library, to MonoGame. As part of this work, we chosen a viable feature subset that is going to be implemented and solved many technical issues. Issues like how to render our graphical primitives or how to implement a non-rectangular clipping. The emphasis of the thesis was to follow the well-known WPF API and its behavior during the implementation process as closely as possible.

Keywords: MonoGame, WPF, Library for creation of user interfaces

Contents

1 Introduction

Nowadays, there is an enormous amount of various computer games. As the developer tools and graphical frameworks are getting more powerful and easier to use, even more developers are getting attracted to this area. This leads to a certain trend that can be observed lately that not only the big game studios are capable of producing successful computer games but there is now also a high number of independent game developers that can achieve the same success. Those developers work either individually or in small teams composed mainly by their friends or other people interested in helping to create a game with a given topic. Those developers are being called the Indie game developers and their games the Indie games respectively. We can mention the games like Minecraft [1] or Terraria [2] as examples of highly successful Indie games.

The process of creating a game can be however very challenging and time consuming. This is especially the case with the previously mentioned Indie game developers whose development resource are limited. By development resources we mean free time, developer experience and the size of their development team. From now on, our focus will be mainly on those Indie game developers.

The first thing a developer needs to do before starting to work on the actual game is to choose a viable graphical framework. We already mentioned that the Indie game developers usually have limited development resources and therefore the framework should be easy to learn and easy to use. One of the popular graphical frameworks among the Indie developers is the MonoGame [3].

MonoGame is a cross-platform and open-source graphical framework that provides to developers access to high-performance graphics. It is trying to achieve the same as DirectX and OpenGL graphical frameworks but while those frameworks are designed to work with the C++ low-level language, the MonoGame provides developers with a convenient managed environment and coding is done in C# or other .NET languages. It is also worth noting that MonoGame is not a project designed from scratch. It is an open-source implementation of the proprietary XNA framework that was originally designed by Microsoft to be used in their Xbox and Windows Phone 7 devices. The situation today is that XNA is no longer in active development and the latest release of the MonoGame – version 3.4 in the time of writing this thesis – claims to be fully API-compatible with this latest XNA release – version 4.0 Refresh in the time of writing this thesis.

However, while the MonoGame does provide the developer with a comprehensive set of graphical capabilities and performance, as it uses DirectX and OpenGL under the hood, it does not implement any sort of out-of-box support for defining user interfaces.

Given to this fact the developer has three possible options how to handle the graphical user interface:

1) Create a custom graphical user interface framework that suits the game's needs from scratch.
2) Render the MonoGame output into some graphical user interface framework that supports this interoperation like the Windows Forms or Windows Presentation Foundation. Alternatively, the other way around, render the Windows Forms or Windows Presentation Foundation into pure MonoGame.

Before we get to the last option, we will examine the two options we just described. The problem with the first option is that such a framework is often not very reusable and will provide serious challenge to adapt to changing needs. As for the second option, the problem with this approach is that this interoperation must be supported on a given platform and could therefore limit the cross-platform support the MonoGame provides. For example, the Windows Presentation Foundation framework is only supported on the Windows platform, which would make the developed application unusable on platforms other than Windows. We can see that neither of these two approaches is the ideal solution for our problem.

3) Finally, the third option is to use some existing library that implements the graphical user interface functionality.

This approach allows the developers to concentrate on game itself and not to worry about the GUI functionality. Moreover, by using a viable library we can also expect it will not suffer the same reusability and adaptability issues as we described for the first option. We will now go through some of the publicly available libraries that are available to use with the MonoGame and show why they are still not the ideal solution we are looking for. The following were picked as examples because they are either still under active development or are at least compatible with the MonoGame/XNA 4. Those examples are taken from the gamedev.stackexchange [4], which is a site where the developers debate about the game development and therefore we can expect that the libraries mentioned there represent some of the most viable libraries to use. The picked examples are: xWinForms [5], Squid [6], NuclearWinter [7], Ruminate [8], and Nuclex Framework [9]. By doing an internet search we can add even more, MonoGame Gui4U [10] and Coherent UI [11]. All these frameworks, with an exception of the Coherent UI, have one thing in common. Each of them comes with its own new API. While some of those libraries might be convenient to use for some developers and for those developers this would be an acceptable solution for the GUI issue, it also brings along it the need to learn a new API.

Coherent UI is an exception as it allows the use of Windows Forms and Windows Presentation Foundation APIs. Unfortunately, this is achieved by using the existing proprietary .NET libraries, therefore limiting the usefulness only for Windows platform and making it unsuitable as a solution for our problem.

However, the idea with providing the developers with access to an existing API is worth pursuing as the developers might prefer to program their GUI using an API they are already familiar with or an API that is more widely spread so they could reuse their existing code in other environments. Therefore, it would be useful to have a GUI library that would implement some already existing and wide spread API for creation of graphical user interfaces and in the same time would be appropriate for use in game development.

In this thesis, we would like to choose such an API and then create a MonoGame library that will implement the missing MonoGame's graphical user interface functionality and will use the selected API as the template.

In the rest of the chapter, we will examine what is expected from every graphical user interface framework and what our library must be capable of doing as well. Then we are going to examine what is important for a user interface library that targets game development and finally we will decide on what GUI framework API we are going to implement in this thesis.

1.1. Graphical User Interface

A Graphical User Interface – or in short GUI – is a mechanism of interaction between the machine and human. User interfaces are usually divided into two types. First one is the Command Line Interface – CLI - and the second one is the already mentioned Graphical User Interface. The difference between the two is that while the CLI uses text output and input to interact with the application user the GUI is using interactive visual objects for this interaction. These visual objects are then interacted with by using various input devices like mouse, touch or keyboard. By visual objects, we mean menus, icons, buttons, sliders, etc. We can see an example of a GUI on the Figure 1:

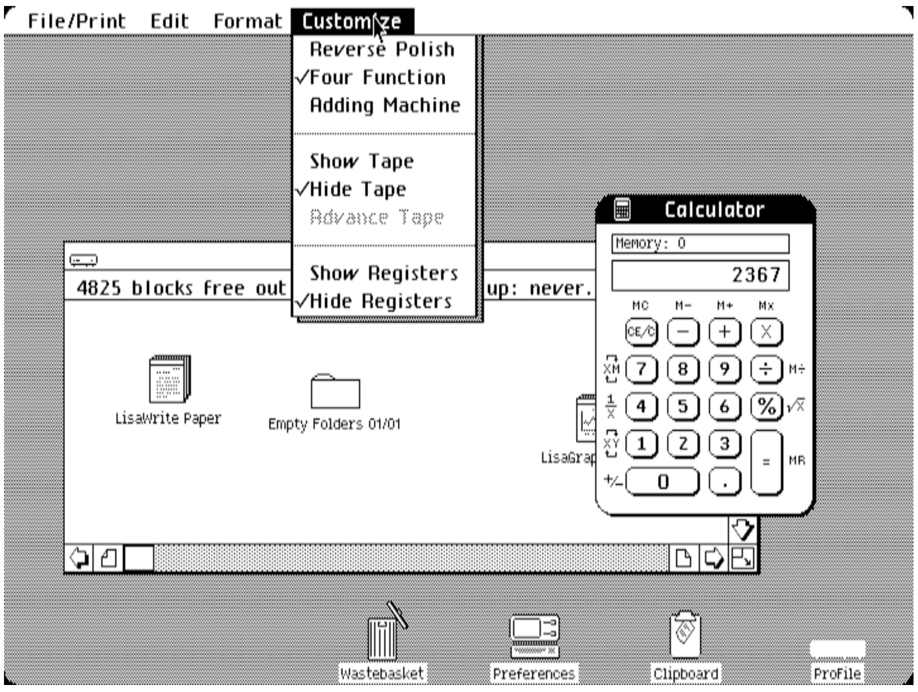


Figure 1: Lisa Office System 3.1 – an example of one of the first GUIs (reprinted from [12])

Rendering

We already stated that a GUI in general contains various visual objects. These visual objects need to be rendered in order for them to appear on the screen. Therefore, our library should be capable of rendering its GUI. The GUI rendering process consists mainly of rendering geometric primitives that in the end form the final GUI. These primitives can also be filled with a color or a texture. For example, if we take yet another look at Figure 1 and decompose these visual objects into a set of geometrical primitives then we can see that this GUI example is composed of primitives like lines, rectangles, rounded rectangles, polygons etc. Moreover, we can see that text is also an important part of GUI and some icons are also present. It is clear that our library should be capable of rendering some of the most common geometrical primitives so they can be used to compose a resulting GUI. In addition to these, we should also support drawing text and images as these two play a big role in the GUI area.

Handling user input

Once we manage to have our GUI rendered, we need to be able to handle user input so the user of the application can actually interact with it. This is why our library needs to implement a mechanism that will provide a feedback every time a user interacts with any of the GUI's elements. This feedback should provide information about which element was interacted with and what kind of user input that was. By user input we mean state changes like mouse button pressed/released, keyboard key presses/releases, etc. These two information are import to the developer so he can correctly decide on how to handle the user input. The most viable and the most widely spread technique to notify about those user inputs is to deploy a so-called event-driven input system. Using this approach, the developers can register only for the types of user input that they want to handle and only on the elements where it makes sense for them.

We already mentioned that the user can interact with the GUI using many different devices. We mentioned the most widely used ones – the mouse, the keyboard and the touch input. In our library, we would like to support at least the mouse and the keyboard based types of inputs.

Stock Controls

We already determined that our library should be able to render itself on to the screen and that it should be able to handle user input for the individual UI elements. Another functionality every GUI framework provides is a set of premade and included elements - so-called controls - that can be used by the developer right out-of-box. Those elements should cover basic GUI usage areas. Included controls usually are: button, controls designed for text input, controls that only present text, image presenters, and various variations of control containers that allow to easily position child controls to form a desired layout for the application. We should include those basic controls in out library as well.

Requirements

Here is the summary of GUI requirements we determined in this chapter:

R1) Our library should be able to render basic geometrical primitives and color or texture them
R2) Our library should provide an event based system of notifications on user input
R3) Our library should include a set of basic GUI controls

1.2. GUIs in games

Now that we have examined the basic features, a GUI library should implement and provide to the developers, the next step is to examine whether there are any specific requirements for GUIs in the world of games. On the Figure 2, we can see an example of game GUI.

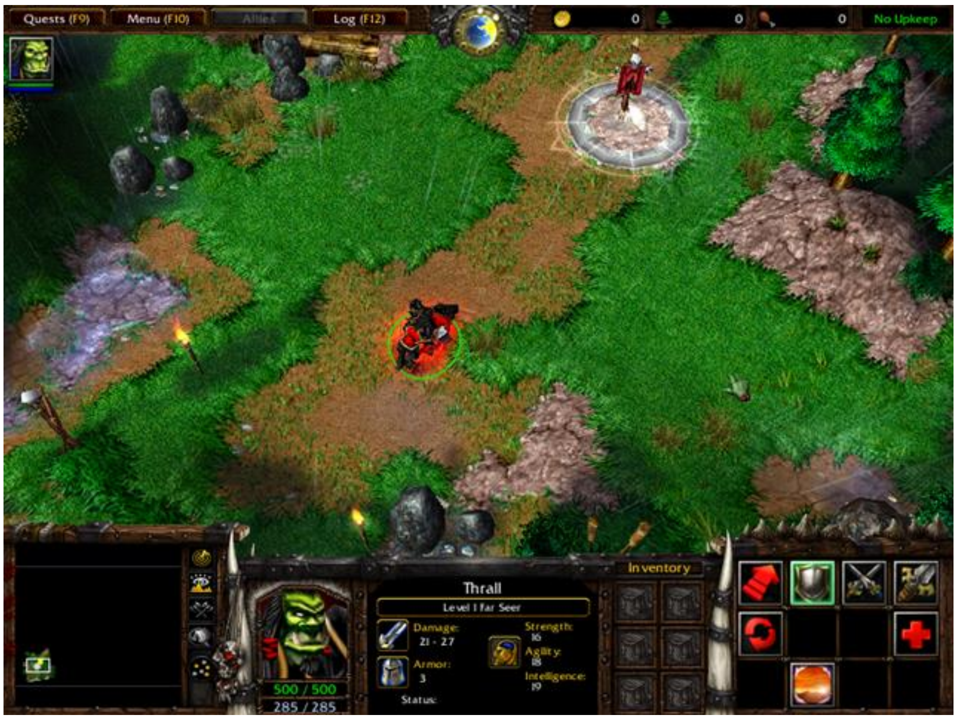


Figure 2: Game GUI example, Wacraft III: Reign of Chaos (reprinted from [13])

On the Figure 2, we can see that games are trying to be visually appealing and to differentiate from one another. What this means for the GUI is that they try to make the GUI look right just for the game they are developing at that moment. So in order to make things easier for the developers our library should provide some level of visual customizability for our controls. Moreover, there might be cases when the developers just want to make a new control all on their own so it fully corresponds to their needs so we should support this custom control creation too.

Requirements

Here is the summary of GUI requirements we determined in this chapter:

R4) Our stock controls should be partially customizable
R5) Our library should provide a way to create a custom GUI controls

1.3. The API choice

So far we already examined the GUI concept and determined what are the basic requirements on our library, now we need to decide on what graphical user interface framework API we are going to use as the API template for our library.

Our requirement was that the selected API should be widely used. Moreover, we target the MonoGame platform, therefore we want to concentrate on the APIs that are being coded using .NET languages. By applying these two conditions, we get the following list of GUI frameworks: Windows Forms, Silverlight, Windows XAML and the Windows Presentation Foundation – in short the WPF.

The Silverlight, Windows XAML and WPF frameworks share many similarities and have a converging API, which is why we are going to count them into one group. Now we need to decide for one representative of this group, whose API we will consider to use as a template for the API of our library. The Silverlight is a framework that is being deployed primarily inside the web pages and lately has been marked as obsolete. Next is the Windows XAML framework. This framework is only available on Windows 8 and newer which is currently limiting its adoption. Finally, the WPF is considered a first class citizen in the area of desktop GUI frameworks, is still under active development and provides support for wide range of Windows operating systems. This makes the WPF the best representative of this category of frameworks.

We are now going to examine the Windows Forms and Windows Presentation Framework, and determine whether any of them complies with all of our requirements (R1) through (R5) as stated in the previous sections 1.1 and 1.2. Then we are going to decide which of these two we are going to use as the API template for our library.

1.3.1. Windows Forms

The Windows Forms is a GUI framework created by Microsoft and included in the .NET Framework since its first release, the version 1.0. It was designed to be a wrapper for the native Windows controls, programmed using the Windows API, so the developers would be able to easily create user interfaces using any managed .NET language.

As a GUI framework, it supports all our requirements as stated in points (R1) through (R3). Let us now examine whether the Windows Forms also comply with our requirements (R4) and (R5).

Windows Forms allows customization of its stock controls by changing the standard properties of controls, such as background color, border width and font size. In the case where this is not enough and a completely different look is needed, it is usually necessary to create a new control that will inherit from the control that is to be visually customized, and override the method that is responsible for drawing the control.

It is also possible to create an entirely custom controls. This achieved by deriving from the `Control` class and overriding appropriate methods that control how the events and painting of the control are handled.

1.3.2. Windows Presentation Foundation

The Windows Presentation Foundation is also a GUI framework created by Microsoft and is included in Microsoft's .NET Framework 3.0 and higher. Most notable features include GPU accelerated GUI rendering using DirectX and the ability to define a custom GUI layout using declarative XAML code. In addition, the developers can easily modify the visual look of existing stock controls as well as the ability to easily create completely new controls. As its basic functionality is the creation of GUIs, we can expect that it is compliant with all our learnt requirements (R1) through (R3) for a GUI framework as described in the section 1.1. Then we just need to examine our requirements (R4) and (R5) as stated in the section 1.2.

In the section 1.2, we determined the requirement that our library needs to allow visual customization of the stock controls (R4). We will now examine how we can customize controls in the WPF.

There are two ways how we can do this. The first way is to alter properties that are already declared on those controls. Using this approach we can alter things like width, height, background, border color and thickness, font color etc. The second way is to define our own Template for that control. This way we can completely redesign the looks of the control because we are altering the individual visual elements the control is composed of. This approach is generally however only used in conjunction with the XAML code. Both approaches are visualized on the following Figure 3:

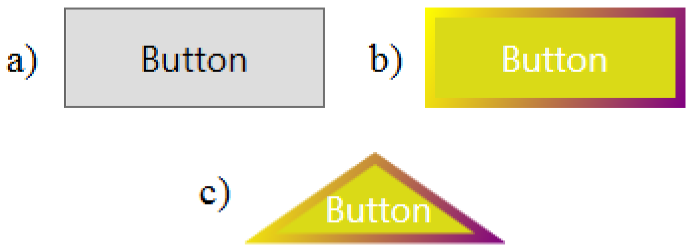


Figure 3: Customizing Button Control

Next to point a) on the Figure 3 we can see the WPF's `Button` Control with its default look and its `Content` property set to the text „Button". Then, next to point b) there is the same control but with several of its properties changed to suit our needs. The properties we changed are namely `Foreground`, `Background`, `BorderBrush` and `BorderThickness`. Finally, next to point c), we can see the `Button` Control with its Template changed. The `Button` does not provide by default any property that controls its shape. Therefore, if we wanted for example a triangular `Button` then we would have to edit the `Button`'s default Template. That is what we did and the result can be seen on the previous Figure 3.

The last requirement we stated in the section 1.2 is that we should allow the developers to easily create a custom GUI elements all by themselves (R5). This is achieved in the WPF by inheriting from either the `UIElement` or the `FrameworkElement` class and overriding the appropriate methods that take care of element's layout and user input handling.

1.3.3. Conclusion

We see that both frameworks fulfill our requirements (R1) through (R5) as stated in both the section 1.1 and the section 1.2. However, the WPF provides a more powerful way of customizing existing controls, and its API is converging with the Windows XAML, which is a part of the new Universal Windows Platform framework. This makes the WPF API a better solution going forward.

Therefore, we decide to use the Windows Presentation Foundation API as the template for our library.

1.4. Thesis goals

In this thesis, we would like to create a GUI library that will enable the developers to easily define a graphical user interfaces in their MonoGame projects. This library will be implemented according to the Windows Presentation Foundation API that we will try to implement as closely as possible. This will not only allow the developers to reuse the existing skills they might already have with the WPF but also to share the GUI code between their MonoGame and WPF projects. No need to learn yet another new API if the developer is already familiar with the WPF. Moreover, we do not want to use any platform-specific functionality while we implement our library, as this would make it harder or even impossible to use this library on other platforms the MonoGame supports. Finally, as the WPF is a very feature rich and complex technology, in this thesis we will implement only a viable subset of the functionality the WPF provides.

Library features

Here we will go over the main features our library should provide.

(G1) MonoGame GUI Library

We want to create a library that will be designed to work with the MonoGame framework and will provide to the developers all the necessary APIs so they can easily create GUIs on the MonoGame platform. In addition, we do not want to use any platform-specific functionality that would make porting our library to other platforms difficult or impossible.

(G2) Rendering

Our library needs to be able to render the GUI as defined by the developer. For this, we need to implement support for drawing various geometrical primitives. We also need to be able to color and texturing them.

(G3) Event-driven user input

The developers needs to be notified whenever user interacts with any control. That is why we are going to implement an event-driven system of notifications.

(G4) Stock controls

We should include stock controls in our library that would cover the basic usage scenarios.

(G5) Controls customizability

To be able to customize a GUI with as little work as possible we are going to implement properties for our controls that would make it possible to change the controls visual appearance. Those properties would be for example able to affect control's background, its border thickness or the border's color.

(G6) Custom controls

When the developers hit the limits of what can be changed about the visual appearance of any of the stock controls then there is going to be a way how they can create their own controls from the scratch with exactly their preferred visual looks and behavior on user input.

(G7) WPF API

Our library will implement a viable subset of the WPF functionality and use its API as a template. In addition, we want our implemented API to resemble the Windows Presentation Foundation's API as close as possible to allow the developers to reuse their code and skills.

Under the hood

Now we will summarize the goals, which are not directly related to the library features.

(G8) Programmed using MonoGame and .NET

The library will be programmed using the MonoGame and Microsoft .NET APIs. As for the programming language, we will use the C# language.

(G9) Operating systems support

For the final version of this library, we want our library to work at least on the Microsoft Windows operating system.

2 Background

In this chapter, we are going to briefly introduce the Windows Presentation Foundation, its basic concepts and some of its features that make it so popular among the GUI developers. We are going to reuse this knowledge in the later chapters.

2.1. Appearance and logic code

The WPF allows the developers to split their GUI appearance definition code and logic code. The GUI code is usually stored in the markup code and the application, while logic is coded in a managed programming language, like the C#.

This approach has the benefit of developing both of these parts independently, which may not only speed up the development but also allow the appearance code to be easily reused in other projects. It also allow designers to work with the appearance code without any extensive knowledge in any particular managed programming language.

XAML Markup

The XAML is a markup language based on the XML language that is used in the WPF for declarative definition of application appearance. Generally, it is used to create and customize various container objects like windows and pages and fill them with content like controls or various geometrical shapes.

On the following Figure 4, we can see an example of a XAML markup code. We define a Window with some specified dimensions and a Grid element as the content of this Window. Finally, we placed a Button element into the Grid and defined a reference to the action that should happen when this Button is click.

```
<Window
x:Class="Markup_CodeBehind.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:Markup_CodeBehind"
mc:Ignorable="d"
Title="Example" Height="100" Width="200">
    <Grid>
        <Button Content="Hello world!" Click="Button_Click" />
    </Grid>
</Window>
```

Figure 4: XAML markup code example

At the run time, the elements defined in this XAML code are converted into instances of WPF objects with attributes set as values of their respective properties.

Such a XAML markup code would produce the user interface we can see on the following Figure 5:

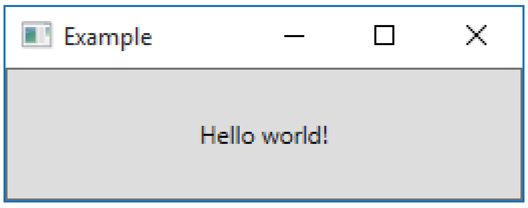


Figure 5: The result of our XAML markup code

Some stylish aspects may vary depending on the Windows version. The example on the Figure 5 is captured on a device running the Windows 10.

Code-Behind

When designing an application, it must be capable of responding to user interactions and perform the expected function. In the WPF, this functionality is implemented in a code that is associated with the XAML markup code. This code is in a form of any managed programming language and we call this the code-behind.

Let us go back to our previous XAML code example. We used a Button element in our XAML code and assigned a reference to the handler that should be invoked when the Button is clicked. This handler is located in our code-behind.

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

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Hello world!");
    }
}
```

Figure 6: Logic code example

We can see that in the constructor method of our `MainWindow class` there is a call to the `InitializeComponent` method. This method is responsible for merging the markup and the logic code of our application. We can now also see the `Button_Click` handler we referenced in our markup code for handling the Button's Click event.

2.2. Controls

Users interact with the so-called "controls". We need to mention that, for example in the Windows Forms this naming would mean that the object derive from the Control class, this is not true for the WPF controls. We can call a "control" every class that visually participates in the user interface and implements some behavior.

Control categories

Included is a rich set of controls that are available for developers to use. List of the controls can be found the MSDN website [14]. Let us mention a few of them and the category they belong to according to their function.

- Input: `Button`, `TextBox`, `RichTextBox`
- Layout: `Border`, `Canvas`, `Grid`, `StackPanel`
- Media: `Image`, `MediaElement`
- Selection: `CheckBox`, `ComboBox`, `ListBox`
- Information: `Label`, `TextBlock`

2.3. Input

Controls must be able to respond to the user interaction. For this, the WPF includes an event-based system of input notifications.

Routed Events

The WPF introduces a concept of so-called routed events. A routed event is a type of event that can invoke handlers on multiple listeners in an element tree, rather than just on the element that raised the event. Such an event can travel the element tree in two directions. Either it can travel from the element that raised the event up through the elements tree or it can travel from the root of the elements tree to the element that raised the event. We call the event traversal that goes the first mentioned way the *Bubbling* routing strategy and we call the event traversal that go the secondly mentioned way the *Tunneling* routing strategy. Finally, we can create a routed event that invokes the handlers only on the element that raised the event. We call this the *Direct* routing strategy.

The following Figure 7 demonstrates this concept. We can see an element, called the “leaf element #2“, that raised a routed event and then we can see how would such an event travel under the *Bubble* and *Tunnel* routing strategies.

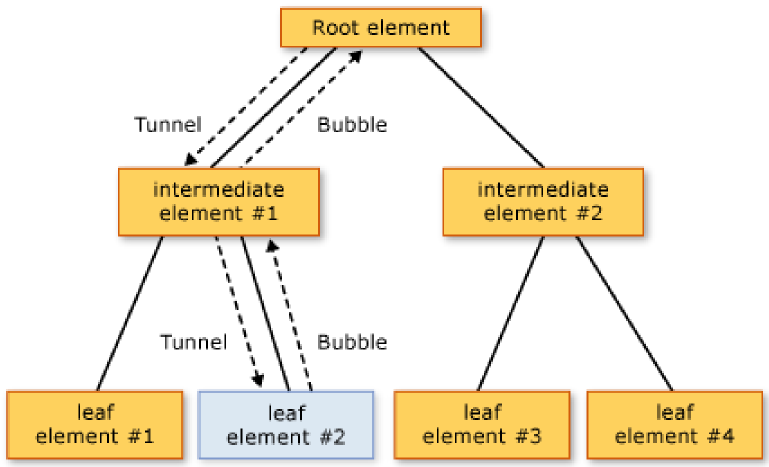


Figure 7: Input Event Bubbling and Tunneling [15]

To handle those events we can either register an event handler for a specific instance of a control or we can register a static class handler that is defined by the class and is also effective on any deriving types. This class handler has the opportunity to handle an event before any registered instance handlers can.

2.4. Layout

Every element is defined by its location and its size. The WPF layout system uses relative positioning for elements to form the desired layout. This process is composed of two stages.

During the first stage, which we call the *Measure* layout pass, the controls are given an information about available space that they can use and their responsibility is to return their desired size, based on this information, back to their parent. Finally, in the second stage, which we call the *Arrange* layout pass, the parents tell their children in what area they are supposed to contain themselves.

2.4.1. UIElement

The core layout functionality is implemented in the `UIElement` class. It defines two public methods to initiate the layout pass on a specified element. These methods are the `Measure` and `Arrange` - where `Measure` initiates the *Measure layout pass* and `Arrange` the *Arrange layout pass*.

Moreover, to allow any deriving class to define its own layout process, the `UIElement` defines the `virtual MeasureCore` and `virtual ArrangeCore` methods that are being called internally as a part of the `Measure` and `Arrange` methods. The implementation for these two methods should also be provided in cases when class that derives from the `UIElement` contains any additional elements.

2.4.2. FrameworkElement

The `FrameworkElement` class derives from the `UIElement` and further extends its layout functionality. The `Margin`, `HorizontalAlignment`, `VerticalAlignment`, `Width` and `Height` are among those new layout capabilities added by the `FrameworkElement`.

These mentioned properties are demonstrated on the following Figure 8. There is a `Grid` control that is the root of the UI. This `Grid` contains a `Border` control (with yellow background) that has its `Margin` property set to position the `Border` approximately somewhere to the middle of the `Grid` control. Finally, the `Button` control is placed inside the `Border` with both the `HorizontalAllignment` and the `VerticalAllignment` properties set to `Center` value, and `Width` along with the `Height` properties set to a custom value.

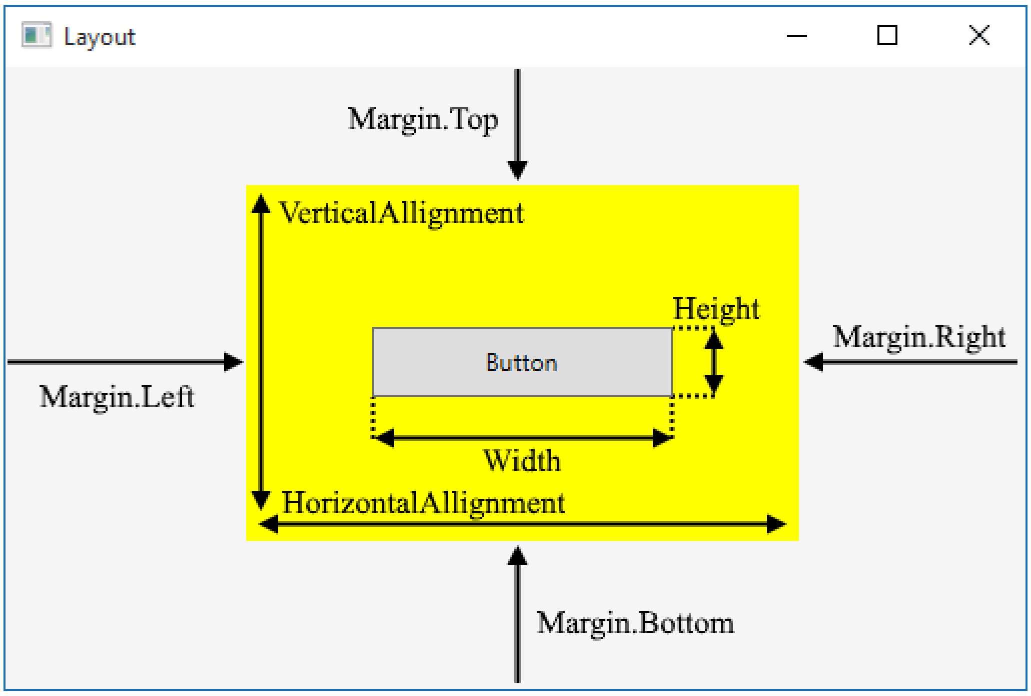


Figure 8: Layout demonstration

The `MeasureCore` and `ArrangeCore` methods are sealed in the `FrameworkElement` as it is the place where the `FrameworkElement` implements this new layout functionality. Instead of these, it provides the `MeasureOverride` and `ArrangeOverride` methods which can be overridden to further customize the layout passes.

2.4.3. Containers

Every container control in the WPF derives from the `FrameworkElement class` either directly or indirectly through other classes. The most common containers in the WPF are the following:

- `Canvas`: Positions its child elements by using coordinates that are relative to the Canvas.
- `Grid`: Positions its child controls into columns and rows.
- `StackPanel`: Positions its child controls either horizontally or vertically one after another.

2.5. Dependency properties

To support some of its advanced features, the WPF uses its own implementation for the property system. We call these properties the dependency properties.

The purpose of dependency properties is to determine the effective value for a property based on the values of other inputs. Those other inputs include the animations, inherited values from parent controls or the mentioned bindings. The full list of inputs and their precedencies is listed in the following Figure 9:

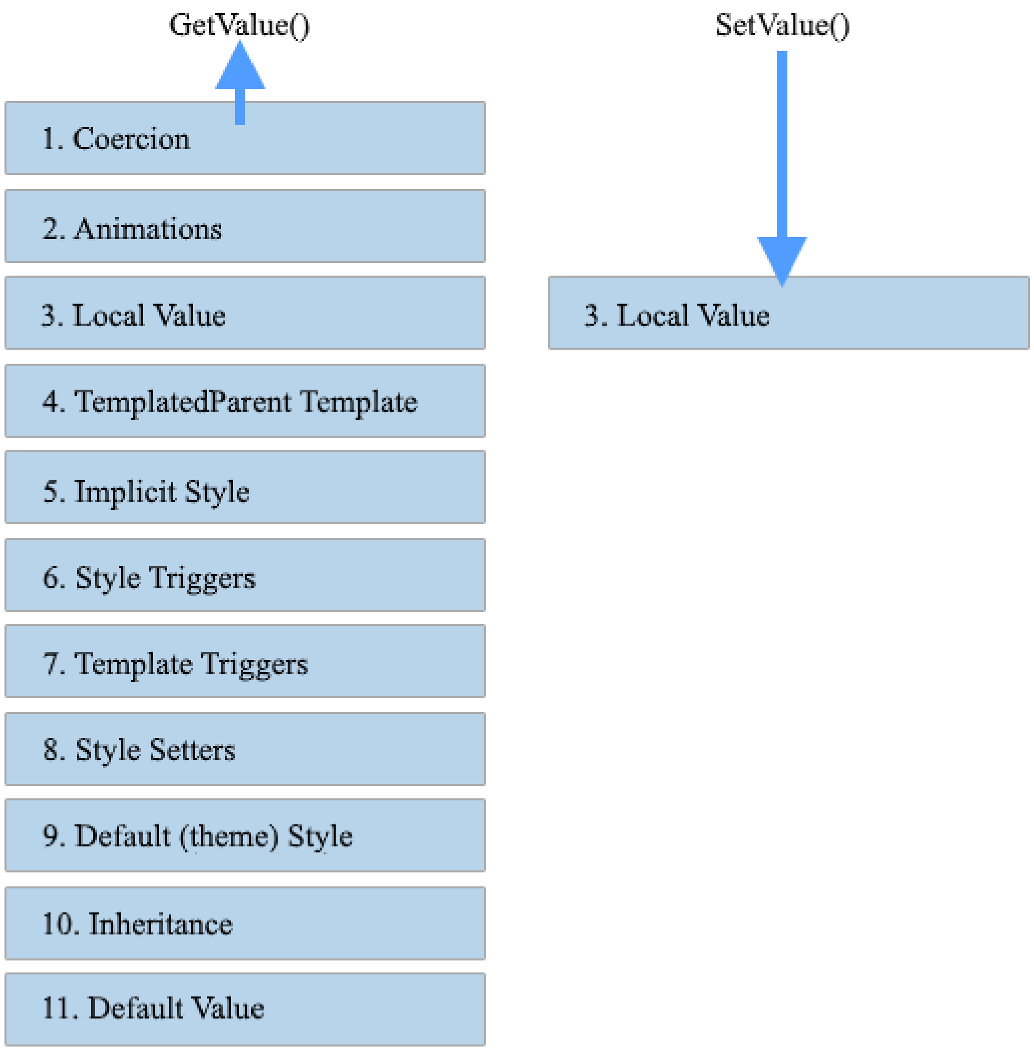


Figure 9: The list of dependency property value precedencies

Moreover, a dependency property can be set up to contain a self-validating mechanism, a pre-defined default value, a callback to notify on any value changes and finally a coerce callback that allows to define a new value for the property based on the actual runtime information. When needed, the deriving classes can change some these characteristics by overriding the dependency property’s metadata.

The dependency properties are usually declared in two steps. Firstly, a dependency property identifier must be declared by calling the static DependencyProperty.Register method. In this method, the property's name, data type, owner type and optionally a metadata and validation callback are defined. Secondly, and this is optional, a new property with both the getter and setter is defined to make the access to this new dependency property more natural. In the getter, the GetValue method is called that takes a dependency property identifier as an argument and returns an object that represents the effective value of the provided dependency property. To set a value for this property, the SetValue method is called inside the setter. This method accepts a dependency property identifier and an object that is to be set as local value for this dependency property. This approach including the optional second step is demonstrated on the following Figure 10:

```
public static readonly DependencyProperty TextProperty =
        DependencyProperty.Register(
                "Text",
                typeof(string),
                typeof(MyType),
                new PropertyMetadata("Hello World!"));

public string Text
{
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
}
```

Figure 10: Declaring a new dependency property

Attached dependency properties

WPF allows setting properties on an object where those properties are not defined. Those properties are called the *Attached dependency properties*.

An example of a usage for the attached property is to allow different child elements to specify unique values for a property that is actually defined on a parent element.

2.5.1. Dependency objects

DependencyObject defines the base class that can register and own a dependency property. It contains a database of locally set values for dependency properties and also declares the two methods mentioned in previous chapter (0) that were used to get and set a value of a dependency property – GetValue and SetValue. To ensure a consistent state of those properties, a cross thread access to those dependency properties is forbidden and this is checked whenever a property is accessed.

2.6. Data Binding

The controls are commonly used to present some data from a data source to the application user. This data is not only presented but also edited. The WPF makes this process easier by introducing a system that takes care about the automatic data synchronization between the source and the target. The core functionality for this is implemented in the `Binding class`.

The binding can be created between any `object` instance on the source side of the binding and any `DependencyObject` instance on the side of the target. However, there is a limitation for the source object. If the source object is not a `DependencyObject` or it does not implement the `INotifyPropertyChanged` interface then it does not notifies the binding system about the bounded property changes. In this case, the binding system synchronizes the value from the source to the target only upon setting the binding and does not reflect any later changes.

There are several modes of binding:

- `OneTime`: Synchronizes the value from the source property to the target property only at a time of applying the binding.
- `OneWay`: Synchronizes the value of the source property to the target property on every source value change.
- `OneWayToSource`: Synchronizes the value of the target property to the source property on every target value change.
- `TwoWay`: Synchronizes the value of the source property and the value of the target property whenever the value of the bound properties changes.

Let us now take a look at the binding example code on the following Figure 11, where we are going to show how to create a binding between the `Label` (displays content) and `TextBox` (accepts text input) controls to automatically synchronize the value of the `TextBox Text` property and the value of the `Label Content` property.

```
// Create controls
Label label = new Label();
TextBox textBox = new TextBox();

// Define new Binding
Binding binding = new Binding();
binding.Path = new PropertyPath("Text");
binding.Mode = BindingMode.OneWay;
binding.Source = textBox;

// Set Binding between the label and textBox
BindingOperations.SetBinding(label, Label.ContentProperty, binding);
```

Figure 11: Binding example code

We firstly created instances of the `Label` and `TextBox` controls, on which we are going to demonstrate the binding. Secondly, we created an instance of the `Binding` `class`, and set the source object and the appropriate path to its property we want to bind. The binding mode was set to `OneWay`, meaning we get the property synchronization from the source to the target. Finally, we bounded these two properties by calling the `BindingOperation.SetBinding` method.

When we now run this example and type “Hello world!” into the `TextBox`, the text gets automatically copied into the `Label`’s `Content` property. We can see the result on the following Figure 12:

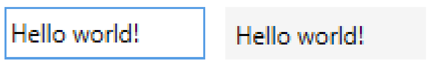


Figure 12: Binding example result

2.7. Graphics

User Interfaces are commonly composed of many graphically different objects. For this purpose, the WPF provides a rich set of graphical capabilities. This is achieved through a full hardware acceleration using the Graphical Processing Unit, in short the GPU. Because of this, the WPF brings native support not only for 2D user interface elements but also supports the use of 3D elements. Moreover, everything is based on vector graphics, which is why the WPF-based user interface will look nice and sharp.

2.7.1. 2D geometrical shapes

The WPF implements support for drawing several two-dimensional graphical primitives. Those primitives are accessible at several layers of the WPF UI system. At the top level, they are all implemented as individual controls. This makes them easy to use and provides them with support for layout and access to the WPF routed event system (because they indirectly derive from the `UIElement` and `FrameworkElement` classes). All those shapes derive from the `Shape class` and can be found in the `System.Windows.Media.Shapes` namespace.

List of Shape objects

Here is a list of shapes that can be found in the `System.Windows.Media.Shapes` namespace:

- `Ellipse`: An ellipse defined by width and height.
- `Line`: A line defined by start point and end point.
- `Path`: A series of connected lines and curves, defined in its Data geometry property.
- `Polygon`: A polygon defined by a set of points that forms a closed shape.
- `Polyline`: A set of connected lines defined by points.
- `Rectangle`: A rectangle defined by width and height. Can be also made round.

Fill and Stroke

Some shapes, where this is applicable, are composed of two visual parts. The fill and the stroke. Fill describes how the interior of the shape should be drawn, while the stroke defines the drawing of the shape's outline. Shapes, for which this is not applicable, like for `Line` or `Polyline`, also define the `Fill` property (as it is declared on the `Shape class` from which all the shapes derive) but effectively use only the `Stroke` property. Therefore, setting the fill property does nothing for those shapes.

This partition of a shape on the fill part and the stroke part allows us to customize those shapes even further by specifying a different visual look for both.

We can see an example on the Figure 13. There are three different looks of the same rounded rectangle. The first rectangle a) is drawn only using the fill, while in the second triangle b) we used only the stroke and no fill. Finally, on the third triangle c) we used both the fill and the stroke.

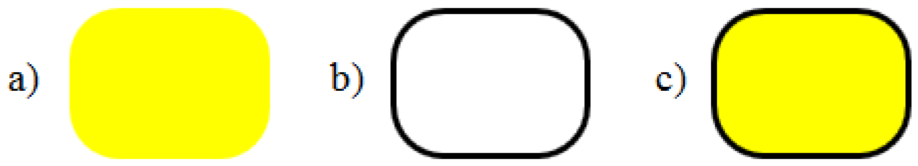


Figure 13: Fill and Stroke example

Moreover, the WPF allows us to customize these shapes even further by altering the default look of the stroke. We have the ability to alter the thickness of the stroke, how it looks on its ends, or even turn the stroke into a set of dashes and many more.

On the Figure 14, we have the rounded rectangle from the Figure 13, however this time we used its `StrokeDashArray` property to create a different looking stroke.

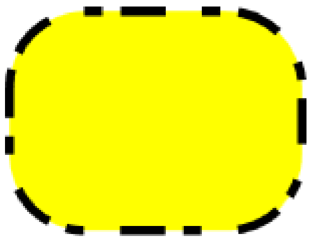


Figure 14: Stroke customization

Brushes

To determine how the area defined by the fill and the stroke is going to be painted we use classes that derive from the `abstract Brush class`.

We can use several brush implementations:

a) `SolidColorBrush`: A brush that paints an area with a simple color.
b) `DrawingBrush`: A brush that paints an area using a `Drawing` object. Such a brush can contain various shapes, images, text and others.
c) `ImageBrush`: A brush that paints an area using an image.
d) `LinearGradientBrush`: A brush that paints an area using a linear gradient.
e) `RadialGradientBrush`: A brush that paints an area using a radial gradient.
f) `VisualBrush`: A brush that paints an area using a `Visual` object.

We can see the example of these brushes a) through f) on the following Figure 15, where we applied these individual brushes to a rectangle as its fill property.

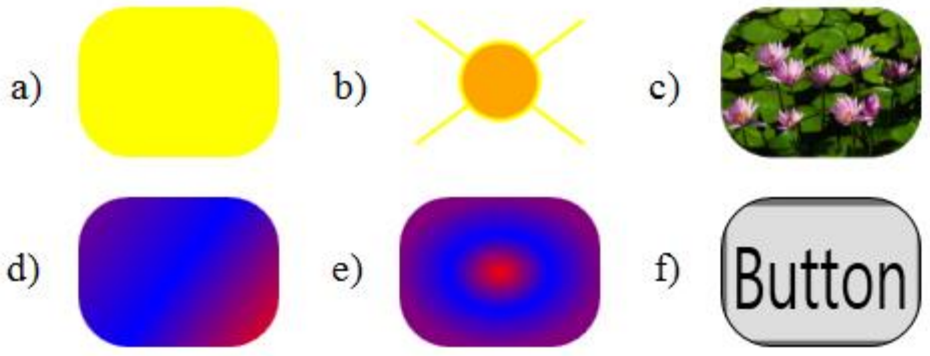


Figure 15: Example of individual brushes applied to Rectangle controls as its fill. Image in c) reprinted from [16]

2.7.2. 3D geometrical shapes

Included in the WPF is also the support for 3D geometrical shapes, which can be mixed with 2D graphics data to create rich controls or provide complex illustrations of data.

To use this functionality, the WPF provides an element called the `Viewport3D` that serves as a “bridge” between the 2D and 3D graphical worlds and can be used as a child of any traditional 2D control.

2.7.3. Visual

The actual rendering capabilities are included in the `Visual class`. The `Visual` therefore serves as a basic building block of every UI element (`UIElement` derives from the `Visual class`). Its main functions are to provide rendering and hit testing support.

In addition to these, the `Visual` also does the following: transformations, clipping, and bounding box calculation. The connection between the rendering and hit testing is very important and leads to a one of the WPF's very useful abilities, and that is a perfect hit testing on individual elements.

Rendering and hit testing

The WPF was designed to take full advantage of the modern GPUs, and is capable of doing all the necessary rendering solely using the GPU. To achieve this it was decided the WPF to be based fully on vector graphics. Therefore, everything the WPF wants to render on the screen needs to be broke down to a set of graphical primitives the GPUs can understand (by those primitives are generally meant the triangles). GPUs can process those primitives at a very high speeds and this makes this kind of rendering quicker than the original way of performing this on the CPUs.

Moreover, the retained rendering mechanism was put in place. Every time we call any WPF's drawing method, our rendering instructions are processed and persisted at the Visual object and then are finally executed at a time the WPF's rendering system sees fit. This opens up not only space for performing rendering optimizations but also allows the WPF to achieve the sooner mentioned feature, the perfect hit testing. It has the description of every rendered geometry and thanks to the retaining system all those information is stored, therefore it is just a matter of simple hit testing those individual primitives.

It is imperative to understand that in the WPF, only that which is drawn to the screen is hit testable. Oppositely, if nothing is drawn, then there is nothing to hit test. This is opposed to for example the Windows Forms, where the connection between rendering data and hit testing is non-existent and only basic rectangular bounding box hit testing is performed.

2.7.4. Clipping

Sometimes it is desired to clip a control (and its children) to make it look nicer and more fitting for the designing user interface or to implement a control that supports scrolling. The WPF supports clipping content with any `Geometry` object that is included in the WPF. Included are many types of defining geometries, like: `LineGeometry`, RectangleGeometry, and EllipseGeometry. On the following Figure 16, we can see an Image control displaying an image while being clipped using the EllipseGeometry.

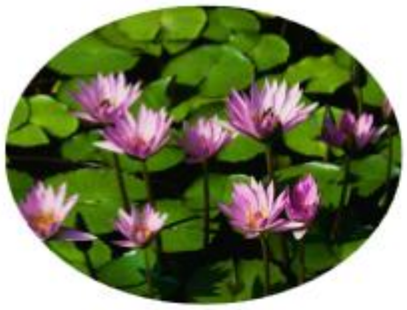


Figure 16: Example of an elliptical clip (reprinted from [16])

2.7.5. Text

To take full advantage of the vector rendering the WPF also interprets and renders all glyphs as a vector graphical data while providing support for *OpenType* font definitions. This process is fully hardware accelerated and the resulting text is also being anti-aliased which results in high performance and high quality, resolution independent text rendering.

2.7.6. Animations

WPF supports a rich set of animations that can be applied to individual properties. Next to this, it also defines animations that can be used to directly change the visual state of a control, those are called the RenderTransforms. Those control animations include the following: scaling, rotating, translating and more.

To animate a property, the following conditions must be met. The animated property must be a dependency property, the object that owns it must implement an IAnimatable interface and the type of the animated property must be supported by the animation system.

2.8. Custom controls

When the stock controls are not enough for a specific task, it is possible to create a custom controls. In WPF, there are three possible models to do this.

2.8.1. UIElement and FrameworkElement model

It was already mentioned in section 2.4 that the UIElement and FrameworkElement actively participate in the layout process and define overridable methods that can be used to change the default layout implementation. In addition to those, the UIElement also defines a virtual OnRender method that can be used to draw graphical shapes. However, when compared to the Shape objects, the shapes drawn during the OnRender method are not individual controls, nor they support any advanced layout and eventing features the `Shapes` objects does. This makes those *OnRender* shapes much more lightweight and should be used instead of the `Shape` objects when creating a complex graphical UI.

DrawingContext

The drawing inside the `OnRender` method is done by using a `DrawingContext` object that is passed into the `OnRender` method as a parameter. This object allows to describe visual content by using draw, push and pop commands.

In the following list are written all the things the `DrawingContext` allows us to draw:

- Drawing: Draws the content of a `Drawing` object.
- Ellipse: Draws an ellipse.
- Geometry: Draws a custom geometry specified by a `Geometry` object.
- GlyphRun: Draws a non-formatted text.
- Image: Draws provided image.
- Line: Draws a line.
- Rectangle: Draws a rectangle.
- Rounded rectangle: Draws a rounded rectangle.
- Text: Draws a formatted text.
- Video: Draws a video.

It also provides other tools that affect the drawings, like transformations, clippings and opacity.

2.8.2. User Control model

When there is a need to reuse a specific set of existing controls more than once in an UI, the easiest way to do that is by deriving from the `UserControl` control and defining all the required controls in this new class. This element tree is then set in the `UserControl`’s `Content` property and every time a new instance of this deriving class is created, all the defined controls, along with their custom functionality, are created as well.

2.8.3. Control model

A new custom element can derive from the `Control class`. This approach is used to build control implementations that allow to separate their behavior from their appearance. This is achieved by using the templates, styles and triggers, which are defined using the XAML markup language.

Most stock controls are using this approach to make their visual customization easier while retaining their original functionality.

3 Problem analysis

In this chapter, we will go through possible realizations and implementations to achieve our goals as stated in points (G1) through (G7). We will break the things we need to decide into three categories. In the first category, we will go through existing open source projects and determine whether we can reuse their code base for our library. Then in the second category, there is the issue of determining a viable feature set of the WPF framework we will implement for our library. Finally, in the last category we will go through some concrete implementation problems and their resolutions.

3.1. Existing projects

As the WPF is a wide spread API, we can assume that there might already exist some efforts for creating an open source and cross-platform implementation of the WPF framework. We should examine such projects as we could reuse some of their codebase in our library. Such approach would bring us several advantages, as we would not have to start coding from scratch and therefore could provide a much more feature rich final version of our library. In addition to the previous, we find a project that is still actively maintained then any new features or fixes released in the new versions of the project could be easily applied to our library as well.

On the other hand, we would like to avoid using any projects that are already abandoned or without any proper documentation. This is due to the fact that such a code would be hard to maintain and to further extend. Finally, we would like to concentrate only on the bigger community project so we have the assurance that the code does not have any undocumented flaws and follows the API properly.

We already stated that the WPF is a part of the Microsoft .NET Framework 3.0 and newer. Therefore, we should start looking for a suitable project at the area of open source implementations of the .NET Framework 3.0 and newer. The most widely used project that meets our criteria is the Mono:

Mono

The Mono [17] is a free and open source implementation of the .NET Framework with support for several platforms including not only Windows but also the OS X, Linux and more. At the time of writing this thesis the Mono project‘s aim is to fully support the .NET 4.5 which is a feature superset of the .NET 3.0. Therefore, we can expect the Mono to provide some kind of implementation for the WPF. Moreover, by looking at the Mono’s source code we can see that it is being programmed in the C# language. However, if we look at the Mono project’s webpage dedicated to the WPF [18] we can see that not only the Mono does not provide any proper WPF implementation but also there are also no plans for one. Stating, “We do not have any plans because the project is too large and there has not been any serious interest from the community to make this effort move forward".

Moonlight

Another area that we should explore are the GUI frameworks that have a converging API with the WPF. The closest matching API we can find is the Silverlight [19]. The Silverlight was created by Microsoft to serve as a cross-browser and cross-platform application framework for creating rich Internet applications. The Silverlight and the WPF are not fully API compatible but there are many things those two have in common and the basic principles are the same. Therefore, we still might be able to reuse some of its code for our purposes. However, as we cannot use the Silverlight's code directly due to its licensing we need to look for a free and open source implementation. A project that provides the closest implementation of the Silverlight framework and is open source is called the Moonlight.

The Moonlight [20] is an open source implementation of the Microsoft's Silverlight framework. Moonlight was primarily developed to provide the ability to run latest Silverlight applications on the Linux and other Unix/X11 based operating systems as the version delivered by Microsoft for these operating systems got outdated and did not implement latest Silverlight features. The Moonlight in its latest release (Moonlight 4 Preview 1) claims to support the Silverlight 4. However, as Microsoft shifted its focus away from the Silverlight framework so did the Moonlight team. Therefore, the Moonlight was abandoned and is no longer maintained with the latest release dating back in the year 2011. By looking closer at the source code, we can see that all the executive code is written in the low-level C++ language and rendering is implemented through using the Cairo [21] graphics library.

However, this project is no longer maintained, which in combination with the fact that it is coded using the C++ language and has no documentation would make it difficult for us not only to add additional features but also to maintain as part of our library.

We see that neither of the considered projects is suitable for our needs and therefore we decide to code our library from scratch.

3.2. Feature set

In the section 3.1 we decided to code out library from scratch. The WPF is a very extensive library with very rich functionality and implementing the entire WPF would be out of the scope for this thesis. Therefore, it is imperative to choose a viable feature subset of the WPF that will be implemented.

This feature subset should cover our goals (G2) through (G6) as stated in the section 1.4. We will now go through these individual goals and we will decide on which functionality of the WPF we are going to implement to achieve those goals.

3.2.1. XAML markup code

The WPF provides a support for defining the user interface appearance using the declarative XAML code that is during the runtime merged with an instance of an object it is bound with. We need to decide whether we are going to implement this support for our library as well.

To support the declarative XAML markup user interface definitions a XAML parser is needed. A proper implementation of such parser is included in the .NET framework, as the WPF itself uses it. Therefore, if we were to support only the Windows platform this would be an acceptable solution and we could use the XAML markup in our library. However, the problem raises on other platforms where only the Mono framework is available as a replacement for the .NET framework. The Mono does not implement a XAML parser that is capable of deserializing the XAML code like the .NET parser does, therefore the functionality of our library would be inconsistent in this area. Moreover, creating our own implementation of the XAML parser would be out of the scope for this work because of how extensive the XAML technology is. And because the XAML is only a supplement for the WPF and our goal (G1) states that we do not want to use any platform specific functionality that would limit our support for other platforms, we are not going to support the XAML user interface declarations on the same level as the WPF does.

3.2.2. Stock controls

Our goal (G4) is to provide a set of controls that can be used right out of the box. As the WPF comes with a large amount of controls and the work on the core functionality of this library is going to take the majority of time, we are going to include only a few controls to cover the basic usage scenarios and to demonstrate the functionality that will be provided by the library.

Among the provided controls should be representatives from the following control categories: input controls, layout controls, media controls, selection controls, and controls that provide information.

Here is the list of these categories and appropriate controls we will implement:

- Input: `Button`, `TextBox`
- Layout: `Canvas`, `Grid`, `StackPanel`
- Media: `Image`
- Selection: `CheckBox`
- Information: `Label`

3.2.3. Stock controls customization

Now we will examine how we are going to handle the support for customizing our stock controls as stated in our goal (G5). In the Introduction chapter, we stated that there are two possible ways how the stock controls can be customized in the WPF.

One way is to set the properties that are defined on those controls and the other way is to use the Templates. Now we need to decide which of these two approaches we are going to implement in our library. Alternatively, if we are going to implement both of them.

The first approach provides the easiest way to modify a control's basic visual look and can be used in both the programming code, like C# code, and the design-centric XAML code. We are going to implement this approach to allow developers basic customization of our stock GUI controls.

As for the second approach, the Template concept is very powerful and useful but it is useful only in conjunction with the design-centric XAML code, as this is the place where the Templates are defined and what makes them so convenient to use. We do not plan any extensive interoperation with the XAML code, as this would be not possible because of the state of available XAML parsers (more in section 3.2.1). Therefore, we are not going to implement the Templates functionality.

3.2.4. Custom controls

It was decided that our library would include a set of controls that can be used by developers right out of the box. Moreover, those controls are also going to be, to some extent, customizable. However, we are creating a library that will be used to create a user interfaces for games. It is necessary to provide a way for the developers to create their own controls from the scratch (our goal G6) so those new controls can be fully visually and logically customized.

There are three different models, as stated in the chapter 2.8, that we are going to consider.

UIElement and FrameworkElement model

First model of creating a custom control is based on inheriting from either the UIElement or the FrameworkElement and overriding the appropriate methods that participate on the layout process. In addition, if there is need to draw some geometrical shapes onto the screen, there is the ability to override the OnRender method.

This model is at very core of the controls implementation in the WPF, therefore we need to implement this model.

User Control model

The User Control model is based on the `UserControl` control. This control simply allows to define a set of children it includes and then reuse this definition multiple times as different objects. This approach is a matter of implementing the `UserControl` control and we are going to implement it.

Control model

The control model is based on the *Templates*, *Styles* and *Triggers*, which are all defined in the XAML markup code. We decided not to build our library around the XAML (in section 3.2.1) and therefore we are not going to implement this model.

3.2.5. Shapes and brushes

In the previous section 3.2.4, we decided that we are going to implement the *UIElement and FrameworkElement model* for creating the user interfaces. These classes use the `DrawingObject` object to draw any geometrical shapes or text to compose their appearance.

The `DrawingContext` supports a wide range of shapes it can draw. We are going to select only a few of those that we are going to support in the first version of our library (our goal G2). We would like to be able to draw at least the following: line, rectangle, rounded rectangle, ellipse, and of course text.

Moreover, the DrawingContext also allows customizing the stroke of shapes by defining the thickness of the stroke and by turning the stroke into a set of dashes. We would like to support the solid stroke with all of our shapes and the dashed stroke at least at the cases of line and rectangle.

Finally, all these geometrical shapes need a definition that will describe how to color or texture them (also part of our goal G2). For this purpose, we need to implement some viable brushes. The most basic brush that we are going to include is the `SolidColorBrush` that fills the shape with one solid color. Next, we need to allow the game developers to use their artwork as the fill for those shapes. That is why we are going to implement the `ImageBrush`. And the last brush we are going to include is the `LinearGradientBrush` that can be easily used to make the user interface more engaging.

3.2.6. Clipping

Every user interface should have the ability to clip controls and their children. This is necessary for implementation of controls that simply need to clip their content, like the `TextBox`, or controls with scrollable areas, like the `ListView`. Therefore, a way to clip content should be included in our library.

We need to decide to what extend we are going to support the customizations of those clipping areas. There are basically two approaches we can take.

First option is to implement a basic rectangular clipping areas. And the second one is to provide a richer set of various geometries that can be used for clipping, much like the WPF does.

We are creating a library that will be used to create user interfaces for games, and we want to provide the game developers with as much ability to customize their user interfaces as possible. Therefore, we are going to implement the second approach and provide support for at least following clipping geometries: line geometry, rectangle geometry, and ellipse geometry.

Lastly, the WPF allows setting the clipping on multiple levels. We would like to support at least the `VisualClip` property of the `Visual` class so the developers can use this while creating their own custom controls.

3.2.7. Data Bindings

In the section 3.2.4, we decided to adopt to User Control model for creating custom controls. The controls created using this approach are composed of other controls, which are however not visible from outside as the entire control acts as a single entity. Whenever any public property of this new custom control is altered, this new value is propagated to a property of an appropriate child and thus updating the appearance (or behavior) of the control.

To make this process more streamlined and automatic, we are going to implement support for data bindings. For the final version of our library, we would like to provide data binding support at least for properties that are defined directly on the level of source and target objects.

3.2.8. Handling user input

Our last goal (G3) we need to go through is to provide an event-driven handling of user input. This is achieved in the WPF with the system of routed events. It is necessary to decide whether we are going to implement this system and to what extent.

In the WPF, it is normal that controls are defined as a composition of several others controls. The concept of routed events allows to handle events that are raised on those child elements directly on their enclosing parents, which makes the creation of new controls more convenient. The same approach of creating controls is going to be possible in our library (more in section 3.2.4). Therefore, we are going to implement the concept of routed events.

Now, there are two different event handlers the developers can register to handle the individual events: *Instance handlers*, and *Class handlers*.

Developers need to be able to register their event handlers for the individual instances of controls, therefore we will implement the support for registering the *Instance handlers*.

Finally, we want to give the developers more control in handling the raised events, therefore, we are going to implement the *Class handlers* as well.

3.3. Technical issues

In the rest of this chapter, we are going to clear some technical problems that are related to the feature subset of the WPF we selected in the previous section 3.2 and examine whether we can provide at least a partial support for the XAML markup code. We are going to break those technical problems into three areas. In the first area, we are going to go through the problems that are connected to the drawing of a user interface visual representation onto the screen (in section 3.4). Then, in the second area, we are going to go through the problems connected to the processing of user input (in section 3.5). And finally, in the last area we will examine whether we can provide at least a partial support for the XAML markup code (in section 3.6).

3.4. Drawing user interface

In this section, we are going to decide on solutions for the problems related to the drawing of our user interface.

3.4.1. Drawing geometrical shapes

Back in the section 3.2.5 we decided on the shapes we would make available in our library. Now we need to decide on the ideal way in which we are going to draw these shapes and how we are going to apply the appropriate color or texture. We are going to consider two different approaches:

a) *Drawing using the CPU*
b) *Drawing using the GPU*

The drawing using the CPU (a), also known as the *software rendering*, is an approach where the CPU is responsible for coloring individual pixels to create a desired shape. This approach has several problems. Firstly, if the shape were to be drawn with a texture, it would be necessary to apply this texture correctly. Since the CPU is a general-purpose processor and the MonoGame targets GPU accelerated game development, there is no existing functionality that could be used and therefore we would have to program this functionality ourselves. Secondly, going through every pixel and calculating whether the shape is defined on the current pixel or not would be CPU intensive. Therefore, limiting the game developers. Finally, this approach would not only be CPU intensive but would also cause often CPU-GPU synchronizations to transfer computed bitmaps from the system memory to the video memory on the GPU so it can be drawn on the screen. As the GPU is stalled until the required synchronization operation is finished, this would also cause GPU performance degradation and therefore drops in the framerates.

The drawing using the GPU (b), also known as *hardware accelerated rendering*, is an approach where the GPU draws on the screen a set of geometrical primitives that together form the desired shape. The GPU is a specialized hardware that is, with its massively parallel computing structure, designed specifically for this kind of workload. It can process the graphics much more efficiently than the CPU and is therefore used to offload this work from the CPU. To take a full advantage of the GPU it is necessary to break down every shape into the set of geometrical primitives the GPU understands. The GPUs used to work with various kind of primitives but lately this list went down only to points, lines and triangles (*DirectX 10+* [2222] , *OpenGL 3.1+* [23]), where the triangles are the most commonly used ones. Using the GPU drawing we can also specify a color or texture coordinate for individual vertices and the GPU takes care about properly applying these definitions on its own. An example of rectangle definition using two triangles (*wireframe*) along with its colored form can be seen on the following Figure 17:

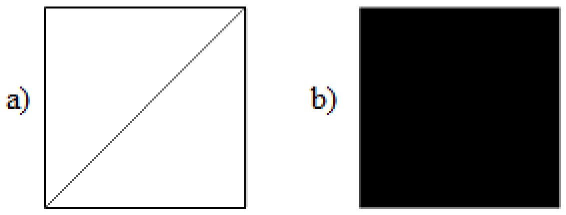


Figure 17: Example of rectangle wireframe and solid form

It can be seen now that the CPU drawing is not the ideal solution for the drawing problem, while the GPU approach covers all our requirements and is more performance efficient. That is why we decide to use the GPU approach. We are also going to use the triangular form of representing our shapes as these primitives are supported on all modern GPUs and are also the fastest ones to draw.

3.4.2. Rendering the fill of graphical shapes

We decided that we would support the drawing of the following four graphical shapes: *Line*, *Rectangle*, *Ellipse*, and *Rounded rectangle*. Now we need to go through these shapes and decide for each how it would be rendered.

However, before we continue with our examination we are going to take a look at some of the possible approaches that are mentioned often on various internet forums and involve using the `SpriteBatch` class. Those approaches include for example drawing a line using a 1x1 texture as described for example on the Game Development Stack Exchange [24], or drawing a rectangle as described on Stack Overflow [25]. By looking at the source code for the `SpriteBatch` class [26] and subsequently `SpriteBatcher` class [27] that is used internally by the `SpriteBatch`, it can be seen that the `SpriteBatch` uses a simple triangulated rectangle onto which it applies the provided texture. We are not going to use any of these approached based on the `SpriteBatch` class because we would like to have full control over the rendering process.

Line

We will start with the *Line* shape, here we are going to consider two possible approaches.

The first approach is to take advantage of the graphical primitives that directly provide support for rendering lines. The MonoGame provides support for the following two types of line primitives: *LineList*, and *LineStrip*. The *LineList* primitive represents a list of isolated, straight-line segments, while the *LineStrip* is a primitive that is composed of connected line segments. Such a line can be easily colored and textured through the exposed properties in the individual vertices of the line (through their `Color` and `TextureCoordinate` properties). However, this line has a fixed thickness of one pixel, and this cannot be altered.

In the second approach, we consider using a triangular representation for our lines. For this purpose, the MonoGame supports the rendering of the following triangle primitives: *TriangleList*, and *TriangleStrip*. The *TriangleList* primitive represents a list of isolated triangles and the *TriangleStrip* is a series of connected triangles. This approach would require us to triangulate each line, which is more work but in return, we can create lines of any thickness. This triangulation can be done by calculating the normals for the two defining points of the line and expanding them outwards by half the thickness on either side. On the following Figure 18 can be seen an illustration of a line geometry created this way:

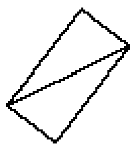


Figure 18: TriangleList based line geometry

The first approach has a serious limitation in the inability to draw lines with custom thickness, which makes it unsuitable for our needs. The second approach provides a way to draw lines with various thickness and also allows us to color or texture them. Therefore, we decide to implement the second approach.

Rectangle

Next shape we need to be able to draw is the *Rectangle*. The rectangle can easily be triangulated into two triangles and drawn either as the *TriangleStrip* or the

TriangleList set of graphical primitives. An illustration of such a geometry can be seen on the following Figure 19:

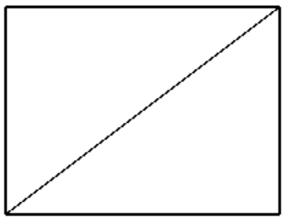


Figure 19: TriangleStrip based rectangle fill geometry

We are going to implement this approach.

Ellipse

For the *Ellipse* shape, we will consider three approaches.

Firstly, we can use a texture of the same size as the ellipse and "cut" it into an ellipse shape by using a custom *Pixel (Fragment) shader*. This pixel shader would receive information about the ellipse, such as its center and size of major and minor axes, and for each pixel, it would compute the ellipse equation and determine whether the pixel belongs to the ellipse or not. If the pixel would belong to the ellipse, the color for the pixel would be sampled from the appropriate position in the input brush texture. Otherwise, the pixel would be set as transparent.

The additional two approaches consider drawing the ellipse using graphical primitives. One approach is to triangulate the ellipse geometry so it could be rendered as a *TriangleList*. This process of triangulation would consist of computing the outline points of the ellipse (by using its *sine* and *cosine* properties and changing the *angle*) and creating individual triangles, which would be composed of: center of the ellipse, lastly computed point, and the current point. An example of resulting geometry of this approach is illustrated on the following Figure 20:

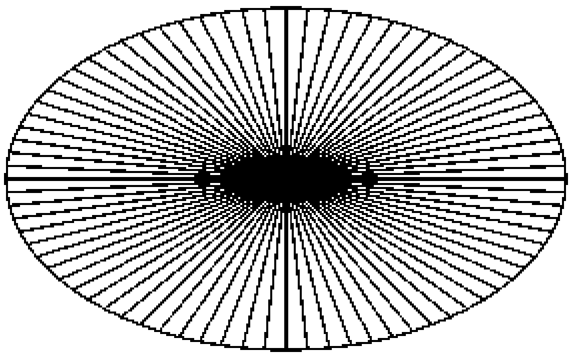


Figure 20: TriangleList based ellipse fill geometry

Finally, there is an article examining the way the WPF does its rendering, *A Critical Deep Dive into the WPF Rendering System* [28]. If we take a look at the way the ellipse is composed we can create a similar approach. Based on this observation, our last approach is to triangulate the ellipse into a *TriangleStrip* geometry. This triangulation would consist of walking down the y-axis of the ellipse (*–minor axis* through +*minor axis*) by one pixel and for each calculating, using the ellipse equation, the appropriate x-axis point. Then by subtracting this calculated point from the x-axis center of the ellipse, we get both x-axis points on the current y-axis level and we add them both to our geometry. Finally, when the walk across the y-axis is finished, the GPU produces very nice representation of an ellipse shape. An illustration of this approach can be observed on the following Figure 21:

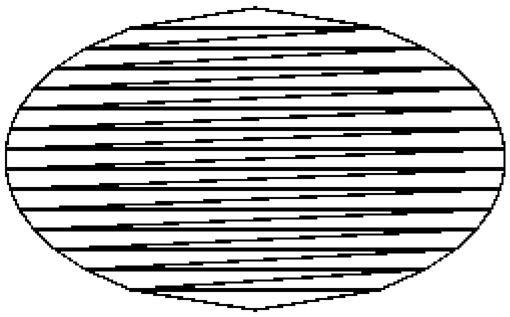


Figure 21: TriangleStrip based ellipse fill geometry

We are going to choose the last approach as it represents a fast and pixel-precise solution, while saving up the memory by using only two vertices per each y-axis pixel.

Rounded rectangle

Lastly, we need to decide on the way, how to draw a *Rounded rectangle*. We are going to consider three possible approaches.

The first one is to draw the rounded rectangle using a *Pixel (Fragment) shader*. This approach is similar to the one we considered for the ellipse, but instead of checking just for one ellipse equation we would be checking for four ellipsis, because every corner of the rounded rectangle is an ellipse with its own center, and space in-between. This would require a lot of branching work (*if-else* statements) in the *Pixel (Fragment) shader*, which would not be very efficient and could limit the use only to newer versions of the *Shader Model* due to instruction limitations imposed by various versions of the *Shader Model* [29].

Secondly, if we break down this shape into more basic shapes, then we see that it is composed of four quarter-ellipses and some rectangles to fill the space between them. To describe this shape to the GPU we can build a *TriangleList* primitive geometry (to make it easy to compose the geometry of different shapes) and triangulate each of the included shapes separately. Where the ellipse parts we would triangulate using the second approach as examined in the *Ellipse* part. An illustration of a geometry created by this approach can be observed on the following Figure 22:

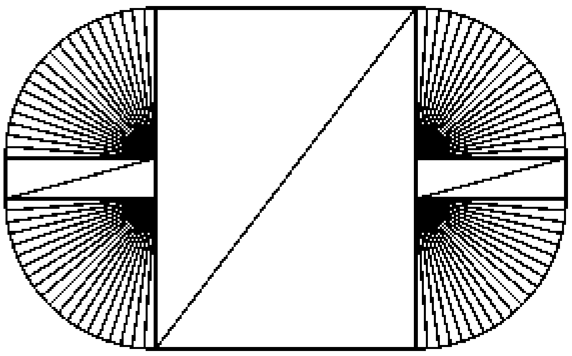


Figure 22: TriangleList based rounded rectangle fill geometry

Thirdly, after observing a rounded rectangle triangulation as pictured on one of the *Microsoft MSDN* pages [30], we can add a third approach, based on the *TriangleStrip* geometry. This approach is very similar to the one we chose for the *Ellipse*. We would begin by precomputing the centers of the four quarter-ellipsis that are a part of the rounded rectangle. And then we would walk down the y-axis and compute the x-axis value by using the ellipse equation. For each y-axis step we would compute two vertices – x-axis point of the top left ellipse center subtracted by the computed x-axis point, and x-axis point of the top right ellipse center added to the computed x-axis point. In both cases, the current y-axis level would be used as y-axis coordinate for the two new points. This would continue until we would reach the height of the provided *minor axis* (`RadiusY` property in the WPF). Then we would continue at the y-axis level computed as the bottom of the rectangle subtracted by the length of the *minor axis*. Finally we would continue the same process of computing vertices with the bottom centers. As we are using the *TriangleStrip* based geometry, the possible free space in-between the top part and the second part is automatically filled by the GPU. On the following Figure 23 we can see an illustration of a geometry created this way:

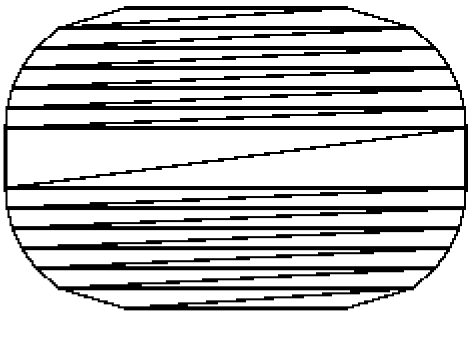


Figure 23: TriangleStrip based rounded rectangle fill geometry

As this shape is not ideal to be processed by the *Pixel (Fragment) shader*, we decide not to use the first approach. The second approach would not allow us to easily build a pixel-precise geometry and its geometry would be also demanding for a number of vertices defining its geometry (because of its *TriangleList* based geometry). Finally, the last approach provides a simple way to compute a pixel-precise geometry and also requires less vertices for its geometry (because of the *TriangleStrip* primitives), that is why we are going to implement the last approach.

3.4.3. Rendering the stroke of graphical shapes

We decided on how to draw the fill of the graphical shapes but most of them also can have a stroke (an outline). We now need to go through the ways how to draw a stroke for the following graphical shapes: *Rectangle*, *Ellipse*, and *Rounded rectangle*. The *Line* is only defined by its fill (`Stroke` property in the WPF), therefore we are not going to make any decisions related to it in this part.

Rectangle

The stroke of a *Rectangle* is composed of straight simple lines, therefore we will use four lines using the *TriangleList* primitives to draw its stroke. An illustration of this approach can be seen on the following Figure 24:

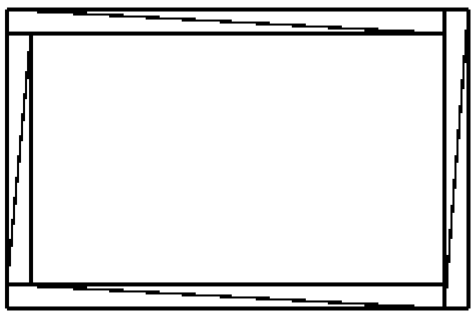


Figure 24: TriangleList based rectangle outline geometry

Ellipse

There are two approaches we are going to consider for an *Ellipse*.

First one is to use our lines to create its stroke. This can be achieved by using the *Bezier cubic curve,* which can give us the individual points of the ellipse outline. However, to create a smooth stroke we would be required to compute a large amount of these lines segments.

For the second approach we will once again look at the *A Critical Deep Dive into the WPF Rendering System* [31] article. By observing the pictured stroke geometry of the ellipse, we can propose the following approach. The stroke of an ellipse can be drawn as a difference between the original ellipse and a new ellipse that is defined with larger major and minor axis (by adding to both the major and minor axis the desired thickness of the stroke). We would be once again walking down the y-axis by one pixel and using the ellipse equation to calculate the x-axis coordinate of both the larger and the original ellipse. Between these two points, we would then draw a line using the *LineList* primitive type. We can see an illustration of this approach on the following Figure 25:

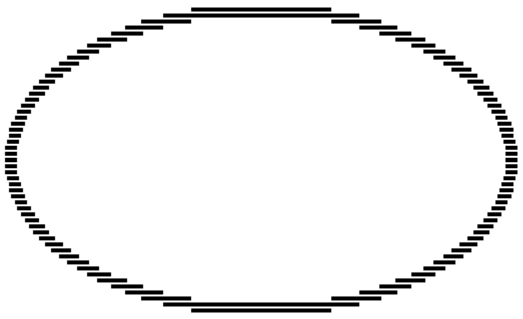


Figure 25: LineList based ellipse outline geometry

The second approach is easier to produce and provides a way to draw a smooth stroke, that is why we decide to use it.

Rounded rectangle

Finally, let us look at the *Rounded rectangle.* The situation here is similar to the one we already examined with the *Ellipse.* We can either draw the stroke by drawing individual line segments or we can build a *LineList* geometry and cover the outline with horizontal lines. The latter would be done similarly to the way we decided for ellipse, however we would be drawing the stroke of four quarter-ellipsis and then filling the space between them. The following Figure 26 illustrates an outline geometry created this way:

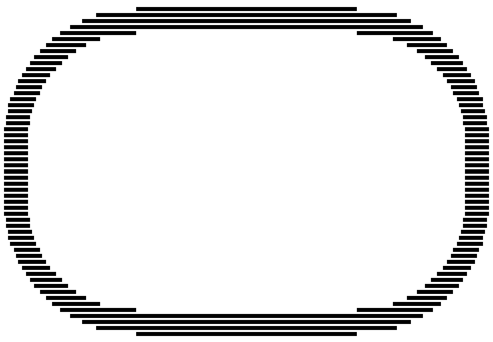


Figure 26: LineList based rounded rectangle outline geometry

As was the case for the ellipse, to provide a pixel precise stroke, we are going to use the second approach.

3.4.4. Vertex buffer

We decided that we are going to represent all of our graphical shapes in the form of a set of some graphical primitives and this approach provides the most optimal way of using the GPU.

However, this computed data, in the form of the vertices describing the individual primitives forming the desired shape, could be stored in a *Vertex Buffer* [24]. This buffer is located in the video memory of the GPU, which makes it the ideal storage for definitions of shapes that are not going to change very often. The use of *Vertex Buffers* requires only the CPU to tell the GPU to draw a certain data it already has, therefore eliminating the times the GPU needs to wait for the data to be transferred from the system memory into the video memory during a draw call and thus further increasing the performance.

We are going to implement this optimization as well.

3.4.5. Text rendering

In the section 3.2.5, it was decided that the text would be one of the graphical primitives we are going to support in our library and in the current section, we decided that we want to draw all our primitives using some graphical primitives and the GPU. However, there is no direct support for handling vector fonts in the MonoGame (meaning the popular *OpenType* and *TrueType* font formats are not directly supported either). When a font is to be used in the MonoGame, it is necessary to provide an already rasterized bitmap of the font character set. This makes it impossible for us to render the individual glyphs as a set of triangles or any other primitive.

This poses several limitations, like the inability to apply a texture as a fill or an outline for a text, or the inability to apply spatial transformations (the resulting quality of the bitmap would be very low). Moreover, even changing the desired font size requires a creation of a new bitmap representation (otherwise the resulting text will become blurry), which is not very convenient. We now need to decide how we are going to solve this font limitation or if we are going to stay with the original bitmap approach.

Firstly, we should consider reusing some existing project that is designed for use with the XNA or the MonoGame and provides the support for drawing vector-based fonts. There is one such a project with said functionality, the Nuclex framework [9]. This is one of the projects that provide support for creation of user interfaces we considered in the Introduction chapter. However, this functionality is deeply integrated into the framework and would require us to include large parts of this framework into our library. We decide against this approach as it could interfere with the workings of our library.

We can also try to implement this support ourselves. This approach would consist of two necessary steps: being able to parse the font definition file, and then being able to render the parsed data correctly. For parsing the font definition file, we can use the FreeType [32] open source library. It is capable of parsing many font definition formats, including the *TrueType* and *OpenType* fonts. Moreover, it provides support for a wide range of operating systems. However, as it is coded using the C language, it is more difficult to use. Fortunately, this issue can be overcame by using the SharpFont library [33] instead. It still uses the FreeType library internally, but provides a wrapper layer for convenient use with any managed .NET language. Finally, once we are able to parse the data, we need to be able to render it. This parsed data usually describe only the outline for individual glyphs. Therefore, in order to be able to fill a glyph, for example with a solid color, we need to triangulate it first. And finally, it is necessary to create a suitable cache store for all these parsed glyph definitions so they do not need to be parsed again, whenever the same glyph is drawn. This approach would be a useful addition to our library but would also require a lot of work and is out of the scope for this thesis.

Because there is not any suitable project that we can use to implement this missing functionality, and the implementation from scratch would be out of the scope for this thesis, we decide to use the original MonoGame font implementation.

3.4.6. Clipping

In the section 3.2.6, it was decided to include support for clipping into our library. In addition, it was also decided that this clipping should be usable with various different geometries. We need to go through possible approaches and pick the one that will suit our requirements.

We are going to consider three possible approaches that can be taken to implement the clipping functionality:

- *Scissor test*
- *Stencil test*
- *Manual clipping*

Firstly, let us take a look at the *Scissor test* [34]. This test usually happens on the GPU right after the *Fragment shader* stage. But there are also cases when this test can happen even before the *Scissor test*, that is as a part of the *Early fragment test* [35]. The boundaries for this test are being set in the window space coordinates (natural pixels) and the processing of the pixels that are located outside of the set *Scissor test* boundaries is stopped, thus clipping occurs. This test should not bring any performance drops as it is fully hardware accelerated and a pixel is only tested for inclusion inside a rectangular geometry. Quite the opposite, using this test can sometimes even increase the rendering performance, as there are some stages of the rendering pipeline that does not need to be executed for that clipped pixel. However, the boundaries for the *Scissor* *test* can be set only using a rectangular geometry, which makes it not a suitable solution for our issue.

Secondly, the *Stencil test* can be used. The *Stencil test* [36] is performed by the GPU right after the *Pixel (Fragment) shader* and *Scissor test* but as is the case with the *Scissor test*, there are cases when the Stencil test can be performed before the *Pixel (Fragment) shader* as well. This test is being run on every pixel that is about to be drawn on and determines whether to allow writing to the pixel or not. If it is determined that the write is not allowed the GPU stops processing that given pixel. This decision is based on a function set by the developer and the current stencil value for the pixel. These stencil values are stored per-pixel in the form of 8-bit unsigned integer value and the storage for all these stencil values is called the *Stencil buffer*.

With the *Stencil test,* it is possible to define custom clipping masks. This is achieved in two steps. Firstly, the stencil function is set to increment the stencil value of every drawn pixel and then the desired geometry of the mask is rendered (with write to color buffer disabled). Secondly, the stencil function is set to keep the stencil value of drawn pixels and to only allow drawing on those pixels that have the desired stencil value. At this stage, the shapes are drawn (write to color buffer enabled) and are automatically clipped by the GPU and the custom stencil mask. This setting of stencil masks is going to be used for all children of a user interface element and they can even define their own clipping masks. Therefore, the actual level (the number of all currently effective clipping masks) of stencil mask must be remembered and this is our desired stencil value we are going to check when the shapes are being drawn. The other way around, when returning from a child with a clipping mask set, the mask geometry must be drawn again with a stencil function set to decrement the stencil value of affected pixels.

An example of this stencil work can be seen on the following Figure 27. Firstly we draw three stencil masks (in an order: rectangular, rectangular, and finally elliptical), each one incrementing the stencil value of a pixel it is drawn on. At this moment, we have three effective masks (Figure 27.a). Finally, we draw a simple rectangle (Figure 27.b) onto the screen, while setting the stencil state to allow drawing on pixels whose stencil value is also three (valid only on those pixels that belong to all our clipping masks). This produces a result in the color buffer that can be seen on the Figure 27.c.

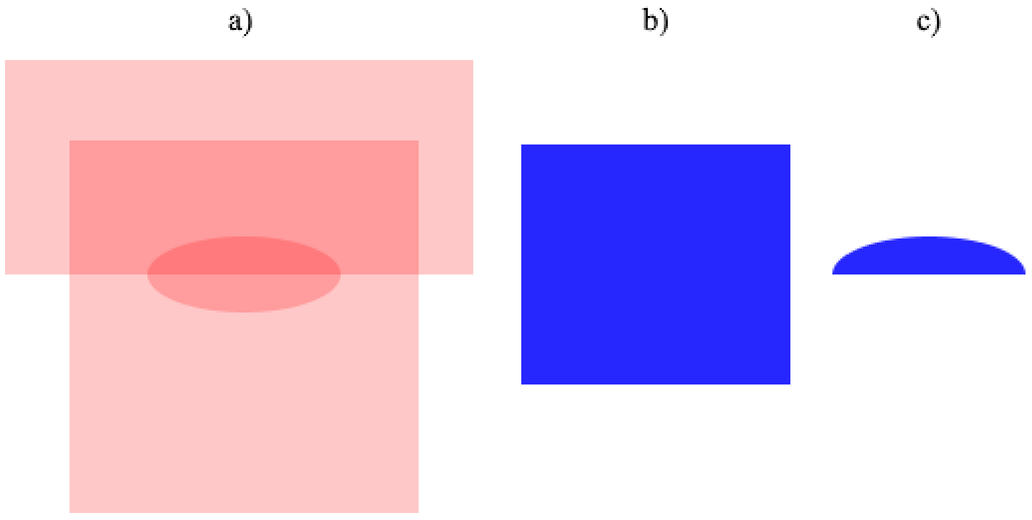


Figure 27: An example of stencil buffer based clipping. In order from left: composed clipping masks in stencil buffer (a), rectangle to draw to color buffer (b), and the result of the stencil clipping after drawing the rectangle to color buffer (c).

As for the performance, the *Stencil test* should have a similar impact as the *Scissor test*, as it is next step in the graphical pipeline right after the *Scissor test* and only an integer value is being checked. However, with the *Stencil test* we are required to make more draw calls as we need to define our stencil clipping mask first and then also clear it.

Lastly, we can do all the clippings manually before we send our geometry to the GPU, determine what parts of the geometry are not supposed to be drawn and remove them. This clipping (intersection of the clipping geometry and a geometry that is set to be rendered) would have to be done for all geometries that are defined on the clipping element itself and on all its children. This would be not only computationally expensive but also nontrivial to implement.

We decide not to use the *Scissor test* as this approach only allows clipping using a rectangular geometry. Both the *Stencil test* and *Manual clipping* allow us to do clip using a custom geometry. From these two we are going to choose the *Stencil test* as it offers the functionality we need while offering a good performance and more intuitive implementation.

However, given to the stencil buffer per pixel size, which is 8-bit unsigned integer, we are limited to 255 nested clipping masks. In any case, this should be more than enough for any real use.

3.4.7. Optimizing the rendering

Finally, we should consider that the game developers usually target approximately 30 to 60 frames per second and they might clear the screen every frame so they can draw a fresh image of the scene. What this means for us is that we need to be ready to redraw our entire user interface for every frame. Such an approach can be however costly on performance, especially if the defined user interface is complex and contains many elements. We need to decide on optimizations so we do not have to completely redraw our user interface for every frame.

Firstly, we are going to examine how is the situation on solely user interface centric frameworks and environments. Those frameworks usually draw their user interface and then keep the user interface representation without a change until they are notified about some actual change in the user interface. In that moment the user interface is redrawn to an updated state. Moreover, this process is usually optimized in a way that only the affected area is redrawn and not the entire UI-containing window.

From the previously stated, we can see two possible optimizations. The first optimization is creating a safe place for our rendered data, so we do not have to render them all over again. This can be achieved by using a texture that we will use as a cache for our user interface. Such a texture would be redrawn only on changes affecting the appearance of the user interface. Then in every frame, we would just draw this cached representation. We are going to implement this optimization. As for the second optimization, it should be implement as well to further increase the performance of our rendering but we are not going to implement it for the first version of our library.

3.5. Processing user input

In this chapter, we are going to decide on solutions for the problems related to the processing of user input, like mouse movement and keyboard key presses.

3.5.1. Handling input

One of our goals (G3) is to have an event-driven system of handling the user input. However, the issue is that the MonoGame does not provide an event based notifications on user input. Instead, the MonoGame provides us on explicit demand with the actual state of input devices, like whether mouse’s left button is pressed or the collection of all the keyboard keys that are pressed at a moment. Therefore, we need to bridge this gap ourselves.

We will solve this problem by watching for changes in the input ourselves. We will do this from inside the MonoGame `Update` method, where every time this method is called we will compare the current state of input devices to the state saved during the last time `Update` method was called. This way we can determine the changes in the user input that happened since the last time the `Update` method was called and then we can raise the appropriate input events on the appropriate controls.

3.5.2. Hit Testing

Another problem that is connected to handling user input and that we need to decide is how we are going to determine whether a particular control was hit on a mouse input. We would like to remain consistent with the behavior of the WPF (our goal G7) and therefore we would like to implement a very precise support for hit testing. Specifically, on the level of actually drawn graphical shapes.

This concept is demonstrated on the following Figure 28. On the figure, we can see a *Visual* object that includes a drawing instruction for a circle. Hit test then succeeds only on positions where this circle is actually being drawn.

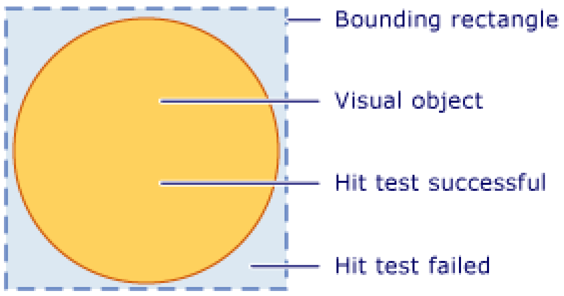


Figure 28: Hit testing drawn content (reprinted from [37])

We now need to go through possible technical implementations and choose the viable one that we are going to use. We are going to consider three different approaches:

a) Hit testing the bounding box
b) Using the *Stencil buffer*
c) Hit testing the drawn geometry

The most basic type of hit testing is by using the bounding box of a control (a). This bounding box is of a rectangular shape and describes the smallest area that contains all the shapes that are drawn as a part of the control. This approach however does not take into consideration our requirement that we want to hit test the individual drawn graphical shapes.

Another way to achieve a perfect hit testing is by using the *Stencil buffer* (b). We can set up the *Stencil test* in a way to write a number one value to all pixels that are affected by our drawn shapes. After this, we would dump the *Stencil buffer* into a bool array and store it with the respective control. This array could be then used to quickly hit test any point by simply looking into the appropriate [x, y] position (as given by the current mouse coordinates) in the array and returning its value. After the *Stencil buffer* would be dumped, it is necessary to clear it so other controls can record their own values. The problem with this approach is that it relies on the ability to dump the *Stencil buffer* and this is not possible in the MonoGame.

Finally, we will go through the last approach (c). In the section 3.4, we decided that we are going to draw our geometrical shapes as a set of lines and triangles. However, instead of disposing of all the computed information about the shapes (after it had been transferred to the *Vertex Buffer*) from the system memory, we can retain this information and use it for the hit testing. This approach would be dependent solely on the CPU and would provide us with a precise hit testing.

We are going to reject the first approach with bounding box hit test as it does not provide the precise hit testing we want. The second approach relies on the ability to dump the stencil buffer. This cannot be achieved using the MonoGame, therefore we cannot use this approach either. Finally, the last approach provides a working way to achieve our goal and allows us to reuse an existing data. That is why we decide to implement this approach.

3.6. Partial XAML markup support

In the section 3.2.1 we decided not to include a full support for the user interface definitions that are written using the XAML markup code. This was because of the current state of the Mono XAML parser. By relying on the functionality provided by the XAML parser, we would introduce a different behavior for our library when run on the Windows and other platforms, which would be in conflict with one of our goals (G1).

There are however developers that want to target only the Windows platform and these developers could benefit from support for the XAML markup. There are some areas where we could provide XAML support and in the same time do not limit our cross-platform support.

Specifically, we are going to support the XAML in three areas:

- Markup attributes
- TypeConverters
- Attached properties

The first area are the markup attributes. Those attributes are used by the XAML parser to better understand the XAML data. They are defined in the `System.Windows.Markup` namespace and to use them we need to reference the `System.Xaml` assembly. This assembly exists in Mono as well and also includes these attributes, therefore we can use it in our library. The attributes we are going to support are:

- `RuntimeNamePropertyAttribute`: Defines which property of a type provides a name for instance of the type. This allows access this object declared in the XAML markup in the code-behind

- `ContentPropertyAttribute`: Defines which property of a type is the XAML content property. This information is used by the XAML parser when processing XAML child elements of the attributed type.

The second area are the TypeConverters. They provide a unified way of converting types of values to other types and are being used by the XAML parser to parse deserialized description of an object in text form and then create the actual instance of that object. The implementation consists of two steps, firstly it is necessary to create a custom `TypeConverter` by inheriting from the `TypeConverter class` and overriding `CanConvertFrom` and `ConvertFrom` methods. In the first method, it is returned that it is possible to convert `string` data and in the second method the actual conversion from `string` is done. Finally, the second step is to connect this custom converter with the actual class. This is done by marking the target class using the `TypeConverterAttribute` attribute and setting the new custom `TypeConverter`. This `TypeConverter` stays in effect for all inheritors until overridden by a different `TypeConverter`. Both the `TypeConverterAttribute` attribute and the `TypeConverter class` are included in the `System.ComponentModel` namespace and the `System` assembly. They are fully supported not only in the .NET framework but in the Mono framework as well, therefore we can use them.

Finally, the third area are the attached properties. During the XAML deserialization the parser has the information about the type of a node it is currently processing based on the node name. With this knowledge, it also knows what properties are defined on that type. However, the attached properties are usually not defined on any other type than on the attached property owner itself. Therefore, the XAML parser needs to know where to look for these attached properties. The *Microsoft Developer Network* website [38] states that, "*The attached property provider must also provide static `Get`**`PropertyName`** and `Set`**`PropertyName`** methods as accessors for the attached property; failing to do this will result in the property system being unable to use your attached property.*". Including these two methods in our codebase for every attached property is going to make them visible for the XAML parser. Implementing these two methods does not require any additional outside functionality and therefore we are going to implement this functionality.

4 Programmer documentation

In this chapter, we are going to describe how the library is connected to the MonoGame, what are the main parts of the library and what are the relations between them.

We used the Microsoft Visual Studio 2015 to develop and compile this library with the addition of the MonoGame libraries, version 3.4.

4.1. Structure of the library

All the functionality implemented in this library is included in one single managed assembly file, called the “MonoGameWPF.dll”.

The implemented functionality itself can be broke down into several areas. On the following Figure 29, we can see the main parts of this library: *Rendering system*, *User input system*, *Property system*, and finally *Control system*.

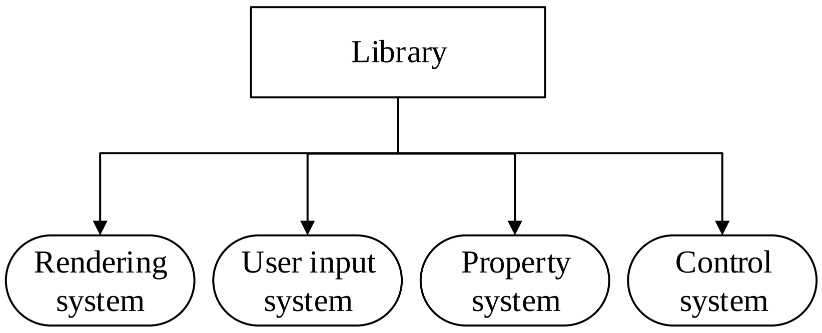


Figure 29: Overview of the main parts of the library

4.2. Overview of the rendering system

The rendering system is responsible for traversing the entire user interface, processing all the rendering instructions any element on the way might have and storing the cached representation of the user interface.

The rendering process is divided into two steps, firstly the entire user interface representation is being rendered into an `RenderTarget2D`. Then in the second step, this representation is being drawn onto the screen.

The library can change the current *Render target* of the graphical device several times during the user interface rendering process (to obtain the texture representations of some brushes), and the user interface needs to be rendered as the last thing on the screen during each frame so it does not get covered by the game itself. This can however affect the rendering of the game (setting the render target might clear the content of the *Back buffer*). Therefore, this two pass process of rendering the user interface has been implemented.

The *pre-render* process can be seen on the following Figure 30. The process starts when the `PreRender` method of an instance of the `PresentationManager` class is called. This `PresentationManager` sets an internal texture as the active *render target*, so it gets the cached representation of the UI and then starts with rendering the specified user interface. This is achieved by walking down the *Visual Tree* and calling on each `Visual` element its internal `Render` method while passing it an instance of the `RenderContext` class. The `Window` class (which also derives from `Visual` class) is used as the root for every user interface and its instance is automatically created within instance of the `PresentationManager` class. Therefore, the rendering process starts there. When the `Render` method is called on an `Window` instance, it automatically calls the `Render` method on its internal storage for rendering instructions, which is implemented as the `RenderDataDrawingContext` class. An instance of this class then goes through all the rendering instructions that are stored within it and renders them. Those individual rendering instructions are represented in a form of individual instances of the `RenderObject` class and are being rendered by calling their `Render` method. Through all these `Render` methods the formerly mentioned `RenderContext` instance is passed. This class provides the necessary methods to draw geometry on the screen and also to affect the rendering process (like setting the opacity and offsets). This process of accessing the render instructions and rendering them is then repeated for all the children of the `Window` instance. The `Render` method is defined as abstract at the `Visual` class level and all the children of the `Window` including the `Window` class itself derive from `UIElement` class (derives from `Visual` class) which provides an actual implementation for this method.

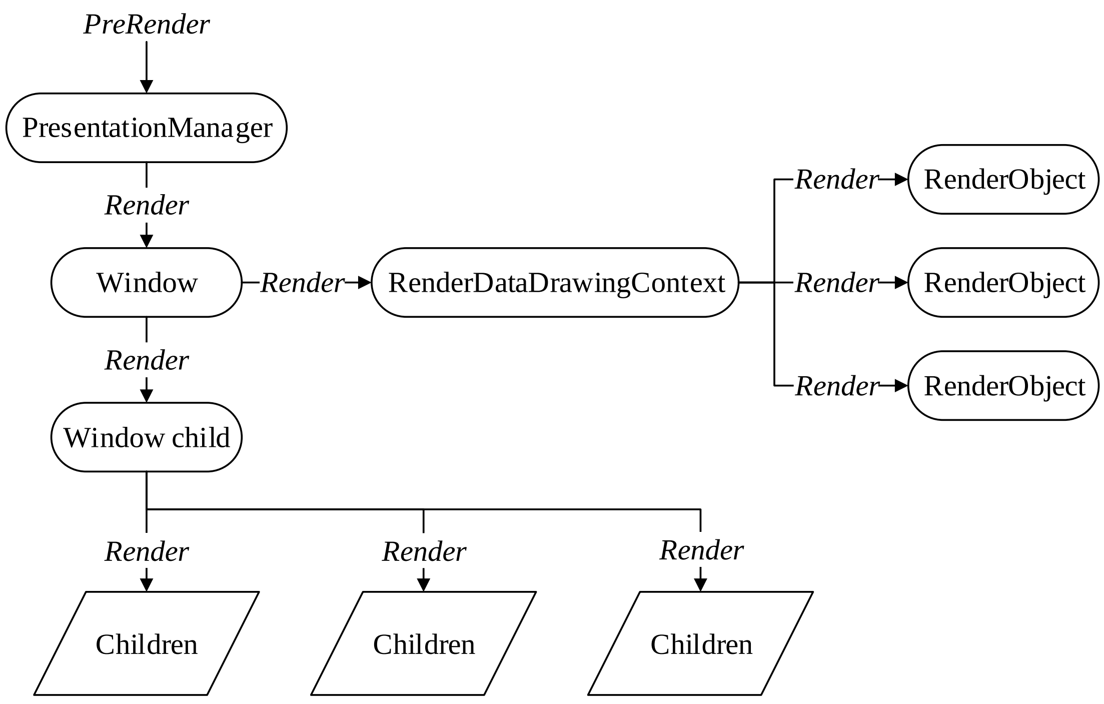


Figure 30: The pre-render process

After the *pre-render* process is finished, the `PresentationManager` instance contains a cached representation of the user interface. At this point, the developers are expected to render the contents of their games on screen. When the developers finish with their own rendering it is finally time to draw the cached representation of the user interface that is cached in the `PresentationManager.` This begins the second stage of the user interface rendering.

In the second stage, the *render* stage, occurs the actual drawing of the user interface onto the screen. The process of this stage is demonstrated on the Figure 31. This process begins when the developers call the `Render` method on the instance of the `PresentationManager` class. There is only one thing the `PresentationManager` does at this point, and that is using the `SpriteBatch` to simply draw the texture of the cached user interface onto the screen.

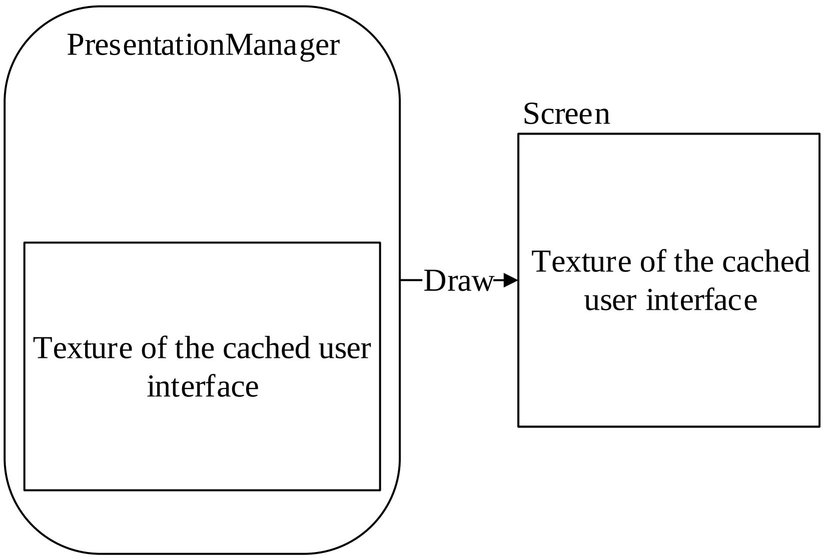

PresentationManager
Texture of the cached user interface
Draw
Screen
Texture of the cached user interface

Figure 31: The render process

4.2.1. Class PresentationManager

The `PresentationManager` class serves as the connection between the user of this library and this library.

During the initialization, this class creates a new instance of the `Dispatcher` class (described in section 4.4.8) and also the `Window` class (described in section 4.2.2) that is then user as the root of the user interface.

This class defines public methods `PreRender` and `Render` that provide the rendering functionality. The `LoadContent` method is used to load the content of our library which is them stored in the static `ResourceDictionary` class from where it is accessible to all part of our library. And the `Update` method that starts the process of updating the user input (more in section 4.3), processing queued `Dispatcher` items (more in section 4.4.8), and processing layout requests (more in section 4.6.2).

The static method `SetWindowTitle` sets a title for current MonoGame client window and the `SetWindowResizeMode` sets whether the MonoGame client window can be resized.

It also contains a storage for the cached representation of the user interface in the form of `RenderTarget2D`.

Class ResourceDictionary

The public static class `ResourceDictionary` servers as a central depository for all external content needed by our library. The storage for this content is implemented as `Dictionary<ResourceType, Dictionary<string, object>>`, where the library currently supports three types of resources. The first kind of resources are instances of the `SpriteFont`. Those are being used for drawing fonts (used by `RenderObjectText` class that will be described in section 4.2.10). The second kind are instances of the `Effect` class. This library uses this to store its shader for *Linear Gradient* (described in the section 4.2.15). Finally, the third kind are instances of the `Texture2D` class.

This class is accessible to third party developers mainly so they can register their own fonts for use in this library and `RenderObjectText` class is using it to access the appropriate `SpriteFont`. However, it is possible to publicly use this storage to store the other two kinds of resources as well.

4.2.2. Class Window

The `Window` class servers as the root of the user interface. It is implemented as any other standard control. We will go through the system of control more deeply in the section 4.6.

The Window class does not include any special functionality except for its `ResizeMode` and `Title` properties. These properties call on value change the static methods of the `PresentationManager` to change the title of the client window and its resize properties.

4.2.3. Class Visual

The abstract `Visual` class provides the basic rendering infrastructure, as seen on the Figure 30. It defines virtual `int VisualChildrenCount` and virtual `Visual GetVisualChild`. The `VisualChildrenCount` is overridden in classes inheriting from the `Visual` class to expose the number of their children. The `GetVisualChild` method is then called to get the actual child. The tree of elements defined this way we call the same as in the WPF – the *Visual Tree*.

The `Visual` class also defines internal `abstract Render` method, which is in our library implemented only by the `UIElement` class (more in section 4.6). This class provides an implementation that calls this `Render` method recursively on every element of the *Visual Tree*.

The storage of rendering instructions for a particular `Visual` is being stored directly in the `Visual` class itself so it can be used by the hit testing mechanism directly on the `Visual` level. This storage is implemented as the `RenderDataDrawingContext` class.

The `Visual` class also defines several properties related to the rendering. Among these properties is the `VisualOffset` property, `VisualOpacity` property, and finally the `VisualClip` property. The `VisualOffset` represents the offset of a `Visual`. In our library it defines the position relative to its parent. The `VisualOpacity` defines the opacity for a `Visual` and the `VisualClip` defines clipping geometry. This geometry is defined using any of the provided classes that derives from the `Geometry` class.

Every `Visual` contains a property identifying its parent in the *Visual Tree*. This property is the `VisualParent` and can be publicly set using protected `AddVisualChild` and `RemoveVisualChild` methods. There are also internal variants of these two methods so classes like `VisualCollection` and `UIElementCollection` can call them and set the `VisualParent` property for newly added children from outside the `Visual`.

Finally, the `Visual` class is also the place where the support for hit testing is implemented. This functionality is provided in the internal `HitTest` method. This method firstly checks whether a point is included in the currently set clipping geometry, and then uses the *depth-first* (and *backtracking*) passage through the `Visual` children while always starting with the `Visual` element that has the highest index (this means it was added lastly and will be rendered on top of all other children). As soon as it determines an element was it, it returns with a result. If none of the children was hit, then the `Visual` hit tests its own content. This hit testing is achieved by using the stored rendering instructions inside the `RenderDataDrawingContext` class. This class exposes a method `HitTest` that checks all the defined rendering instructions for a hit. This default behavior can be changed by overriding the virtual `HitTestCore` method.

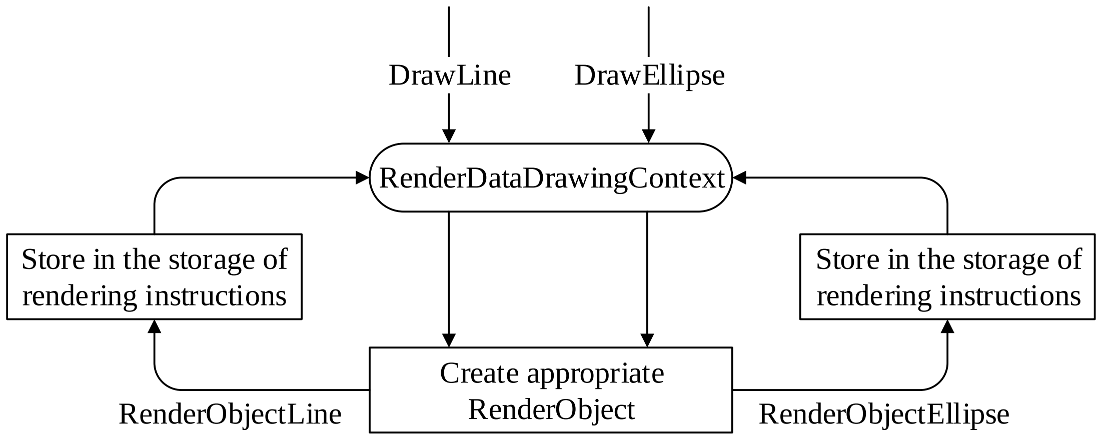
Furthermore, to optimize this process of hit testing, each `Visual` stores a cached bounding box of its content and the content of all its children. Therefore before the `HitTest` method of the `Visual` dives deeper into *Visual Tree*, it firstly checks for this cached bounding box and determines whether that part of the *Visual Tree* can contain the given point or not. The bounding box for a given `Visual` is determined by its rendering instructions and is exposed as the `ContentBoundingBox` property of the `RenderDataDrawingContext` class.

The caching of the bounding boxes is implemented in the following private methods: `CacheVisualBounds`, `RecomputeBoundsDescendants`, and `RecomputeBoundsAncestors`. This process basically consists of getting the bounding boxes for the content of individual `Visuals` and on those applying a union operation.

4.2.4. Class RenderDataDrawingContext

The internal class `RenderDataDrawingContext` serves as a storage for rendering instructions. These instruction can be hit tested and also rendered.

The following Figure 32 demonstrates the process of defining a new rendering instruction. The rendering instructions are stored in the `List<RenderObject>` and are added using various methods, each for a specific type of geometry. Currently implemented public methods are: `DrawEllipse`, `DrawImage`, `DrawLine`, `DrawRectangle`, `DrawRoundedRectangle`, `DrawText`, `PushOpacity`, and `PopOpacity`. These methods create an appropriate variant of the `RenderObject` and store it to the storage of the rendering instructions.



DrawLine
DrawEllipse
RenderDataDrawingContext
Store in the storage of rendering instructions
Store in the storage of rendering instructions
Create appropriate RenderObject
RenderObjectLine
RenderObjectEllipse

Figure 32: Process of creating a new rendering instruction

To keep everything around rendering unified, whenever a `VisualClip` property of the `Visual` element is changed the `SetClippingGeometry` method of the `RenderDataDrawingContext` is called and provided with the specified `Geometry`. According to the provided `Geometry` a representation in a form of `RenderObject` is created and then saved in the public `ClippingGeometry` property. This `RenderObject` is then also used by the `Visual` class to determine whether a point belongs to its clipping geometry.

4.2.5. Class RenderObject

The internal `RenderObject class` is an abstract class that defines a unified way for rendering all the graphical primitives, including their hit testing, and exposing their containing box bounds.

Every inheritor is expected to provide an implementation for the following two abstract methods: `Render` and `HitTest`. The first method is called to render the graphical primitive and is provided with an instance of `RenderContext`, while the latter is used to hit test the geometry of the primitive for a point inclusion.

The `RenderObject` also provides several support (`protected`) methods to its inheritors. The methods `DetermineFillBrush(Brush)` and `DetermineStrokeBrush(Brush)` are used to determine whether the provided brush requires getting a defining texture or it simply provides a solid color that should be used to color the individual vertices. This information is then stored in the `IsStrokeTextured` and `IsFillTextured` `bool` properties. The method `bool HitTestTriangleListGeometry` hit tests a point against a provided `VertexPositionColorTexture` array (passed as a reference because the geometries can get complex, thus saving the unnecessary memory allocations). This method is for testing a geometry defined as *TriangleList* primitive. There are also methods designed to test the *TriangleStrip* and *LineList* geometries, the `HitTestTriangleStripGeometry` and the `HitTestLineListGeometry`. Finally, the last protected helper method is the `VertexBuffer StoreVertexDataColorTexture`. This method receives an array of `VertexPositionColorTexture`, which defines a geometry for some primitive, and stores this geometry in the *Vertex Buffer* that is then returned to the caller.

Lastly, because the `RenderObject` inheritors can contain an instances of the `VertexBuffer`, which is an unmanaged data storage on the GPU, it is imperative to have an infrastructure in place that will make sure all its data are safely disposed. For this purpose, the `RenderObject` implements the `IDisposable interface` and the *Disposable pattern* [39].

HitTestHelper class

The internal static `HitTestHelper` class contains two methods that are used by the `RenderObject` class for hit testing.

First is the `bool IsTriangleHit`. This method is responsible for determining whether a provided point is inside of triangle. For optimal performance, all the parameters for this method are passed as references (`ref` keyword) because they are not being modified and the calls to this method happen very often. The triangle hit testing itself is performed by checking the triangular *Barycentric coordinates*. The code implementation for this method has been adopted from the following source [40].

The second is the `bool IsLineHit`. This method is used to hit test lines. Like the previous one accesses the provided data using a reference for performance reasons. The hit testing for a point inclusion on a line is implemented using standard *Two-Point form* [41] of the line equation.

4.2.6. Class RenderObjectLine

The internal `class RenderObjectLine` provides an implementation for the abstract `RenderObject` class and serves as the means to compute, render and hit test the line graphical primitive.

When an instance of the `RenderObjectLine` class is created, it is provided with several information. These are the line defining `Pen` object, start point of the line and the end point of the line. All this information is used to triangulate the final line geometry. For this purpose, the `ComputeStrokeRenderingData` method is called.

This method either creates a geometry for one solid line (in case when the `Pen.DashStyle.Dashes` is empty) or creates a geometry for a dashed line. To produce this kind of line the equation for *Linear Bezier curve* [42] is used. Moreover, both geometries are generated as the *TriangleList* primitives.

On the following Figure 33, we can see a demonstrational wireframe for the computed geometry of line:

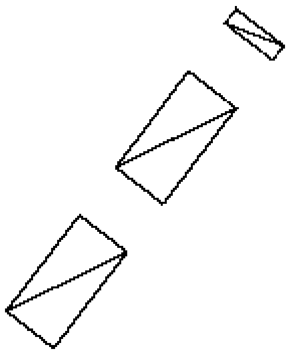


Figure 33: The stroke geometry of a line

This computed geometry is then stored into a `VertexBuffer` by calling `StoreVertexDataColorTexture` and when requested in the `Render(RenderContext)` method, rendered either by calling the `RenderContext.RenderGeometryTextureFromVertexBuffer` or the `RenderContext.RenderGeometryColorFromVertexBuffer` depending on `Brush` (`Brush` will be described later in section 4.2.13) defined.

4.2.7. Class RenderObjectEllipse

The internal `class RenderObjectEllipse` is another implementation for the `RenderObject` class and provides the support for rendering the ellipse geometrical shape.

There are two types of geometries that could be computed for the ellipse primitive. The geometry defining its fill and the geometry defining its outline. The fill geometry is computed by the `ComputeFillRenderingData` method, while for the stroke the `ComputeStrokeRenderingData` is used. These methods are concrete implementations for the triangulation approaches decided in the Analysis chapter (sections 3.4.2 and 3.4.3).

On the following Figure 34, we can see a demonstrational wireframe for the computed geometry of an ellipse:

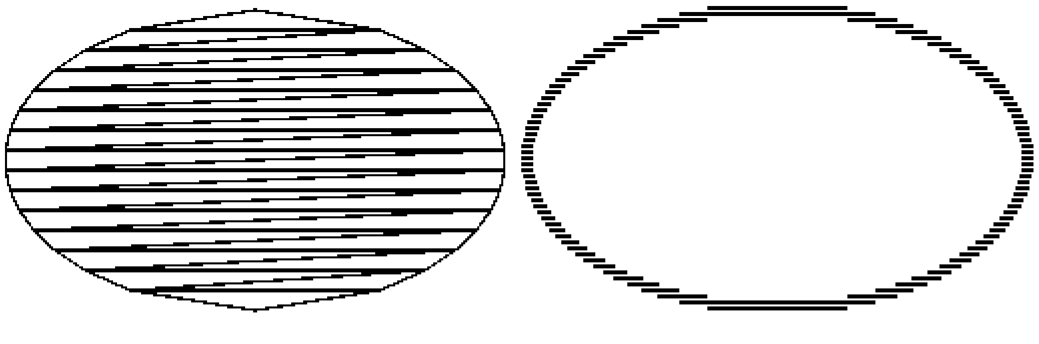


Figure 34: The fill and stroke geometry of an ellipse

The fill geometry is generated in the form of *TriangleStrip* primitives, while the outline geometry is in form of *LineList* primitives. The image of both these geometries was taken with lower geometry quality, so their structure can be actually observed. That is why the ellipse on the left part of Figure 34 looks deformed near its top and bottom.

This computed geometry is then stored into a `VertexBuffer` by calling `StoreVertexDataColorTexture` and when requested in the `Render(RenderContext)` method, rendered either by calling the `RenderContext .RenderGeometryTextureFromVertexBuffer` or the `RenderContext .RenderGeometryColorFromVertexBuffer` depending on `Brush` (`Brush` will be described later in section 4.2.13) defined.

4.2.8. Class RenderObjectRectangle

The internal `class RenderObjectRectangle` is an implementation for the `RenderObject` class that provides the support for rendering the rectangle geometrical shape.

The `RenderObjectRectangle` can have two different geometries. One for the fill and the second for the outline. The fill of the rectangle is computed simply by dissolving the rectangle into two triangles using the *TriangleStrip* primitives, while for the outline is used the same approach as for the line primitive in `RenderObjectLine`. However, as the outline should be textured as a whole with a texture of the same size as is the rectangle, the process of defining the texture coordinates differs from the one used in `RenderObjectLine`.

On the following Figure 35, we can see a demonstrational wireframe for the computed geometry of a rectangle:

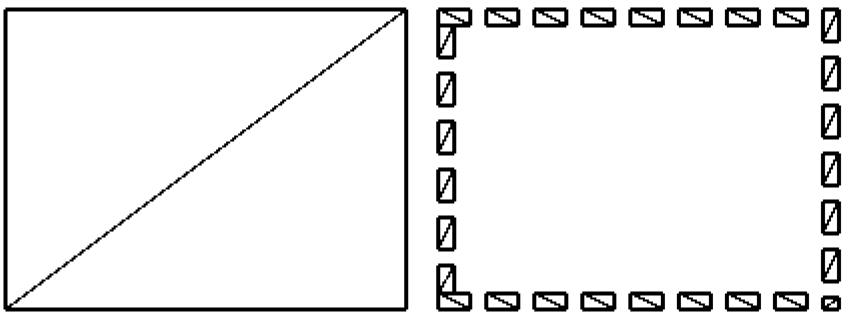


Figure 35: The fill and dashed stroke geometry of a rectangle

This computed geometry is then stored into a `VertexBuffer` by calling `StoreVertexDataColorTexture` and when requested in the `Render(RenderContext)` method, rendered either by calling the `RenderContext .RenderGeometryTextureFromVertexBuffer` or the `RenderContext .RenderGeometryColorFromVertexBuffer` depending on `Brush` (`Brush` will be described later in section 4.2.13) defined.

4.2.9. Class RenderObjectRoundedRectangle

The internal class `RenderObjectRoundedRectangle` provides an implementation of the `RenderObject` that supports the drawing of the rounded rectangle geometrical shapes.

For the rounded rectangles, there is support for computing and drawing fill geometry and full stroke geometry. The fill geometry triangulation is implemented in the private `ComputeFillRenderingData` method and stroke triangulation in the private `ComputeStrokeRenderingData` method. These methods are concrete implementations for the triangulation approaches decided in the Analysis chapter (sections 3.4.2 and 3.4.3).

On the following Figure 36, we can see a demonstrational wireframe for the computed geometry of a rounded rectangle as defined in (e) and a stroke geometry computed for the same shape:

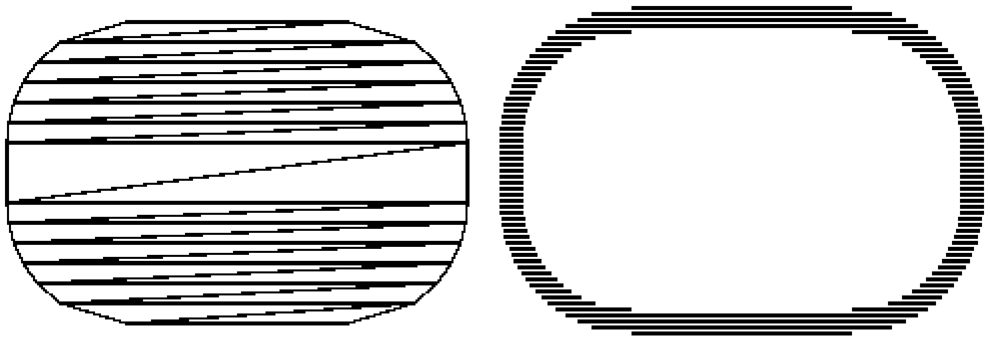


Figure 36: The fill and stroke geometry of a rounded rectangle

The fill geometry is generated in the form of *TriangleStrip* primitives, while the outline geometry is in form of *LineList* primitives. The image of both these geometries was taken with lower geometry quality, so their structure can be actually observed. That is why the rounded rectangle on the left part of Figure 36 looks deformed near its top and bottom.

This computed geometry is then stored into a `VertexBuffer` by calling `StoreVertexDataColorTexture` and when requested in the `Render(RenderContext)` method, rendered either by calling the `RenderContext .RenderGeometryTextureFromVertexBuffer` or the `RenderContext .RenderGeometryColorFromVertexBuffer` depending on `Brush` (`Brush` will be described later in section 4.2.13) defined.

4.2.10. Class RenderObjectText

The internal class `RenderObjectText` provides an implementation of the `RenderObject` that supports the drawing of text.

This class uses the `ResourceDictionary` (described as part of the section 4.2.1) to get the proper `SpriteFont` as defined by given font (`string`) name. Finally, when asked to render itself during its `Render(RenderContext)` it calls the `RenderText` method of the provided `RenderContext` instance to actually draw the text on the screen.

4.2.11. Class Geometry

The abstract `class Geometry` is implemented solely to provide a consistent API with the WPF in the area of `Visual.VisualClip` property.

Deriving classes of the `Geometry` class store basic information that describe a certain geometry. This information is then used by the `RenderDataDrawingContext` class to construct an appropriate `RenderObject`. There are currently three implementations for the `Geometry` class, each representing one of the currently implemented `RenderObjects`.

Class LineGeometry

The `LineGeometry` class stores information about a line geometry. It is used by the `RenderDataDrawingContext` class to produce a `RenderObjectLine` object.

Class RectangleGeometry

The `RectangleGeometry` class stores information about a rectangular geometry (including the corner roundness). It is used by the `RenderDataDrawingContext` class to produce either a `RenderObjectRectangle` or a `RenderObjectRoundedRectangle` object.

Class EllipseGeometry

Finally, the `EllipseGeometry` class stores information about an elliptical geometry. It is used by the `RenderDataDrawingContext` class to produce a `RenderObjectEllipse`.

4.2.12. Class RenderContext

Let us now look at the class that is being referenced through all these `Render` methods.

The internal `RenderContext` class is at the very core of all rendering functionality. When an instance of the `RenderContext` is created it creates several resources that are then being used for rendering. Among these resources are two instances of `BasicEffect` class (which are basically stock shaders). One is set up to be used for rendering geometry with texture and the other one for rendering using the color defined in vertices. Then three instances of `DepthStencilState` are created. Each one is used for a different type of rendering. One is used to set up a new stencil mask, another one to render user interface geometry while allowing to write only where the stencil buffer has the needed value and finally the last is used to clear a stencil mask. Finally two `BlendState` instances are defined. One is used while rendering the geometry of a user interface and the second one has writes to *Color buffer* disable and is used while rendering stencil mask.

The public `PrepareForRendering` method is used by the `PresentationManager` before starting the *Pre-Render* stage. During this method the `RenderContext` makes sure it clears its content from the previous Visual Tree passage, sets the appropriate `BlendState` and `DepthStencilState` to the `GraphicsDevice` and computes a new *Projection matrix* in case the resolution changed.

Then there are several public methods that are being used during the Visual Tree traversal to affect the position of rendered geometry, its clipping, and opacity. These methods are `PushOffset`, `PopOffset`, `PushClip`, `PopClip`, `PushOpacity`, and `PopOpacity`.

There are four available rendering methods then are being used by the individual `RenderObject` instances, in their `Render` methods, to render a textured or colored geometry. All these methods are provided with a `VertexBuffer` instance representing the geometry and the `PrimitiveType`.

The public `RenderGeometryTextureFromVertexBuffer` has two overloads. The first one takes as a parameter an instance of `Brush`. This method firstly creates a `Texture2D` representation of the provided `Brush` by calling the `Texture2D Brush.GetTexture` method and calls the second overload of this method while providing it the created `Texture2D`. This method then sets up the `BasicEffect` shader to use the provided texture by calling the private `PrepareTextureRendering` method. During this method all the necessary information are stored in the texturing `BasicEffect`. Among these information is a *Translation matrix* that is created based on the current offset (set up by `PushOffset` and `PopOffset` methods), current opacity (set up by `PushOpacity` and `PopOpacity` methods), and finally the texture. Finally, the `RenderGeometryTextureFromVertexBuffer` method calls the private `RenderGeometryFromVertexBuffer` method.

The public `RenderGeometryColorFromVertexBuffer` method is used to render colored geometry. It firstly calls the `PrepareColorRendering` that prepares the coloring `BasicEffect` shader and then calls the `RenderGeometryFromVertexBuffer` method.

The last rendering method exposed to any `RenderObject` is the `RenderText`. This method renders the provided text, using the provided `SpriteFont` through calling the `SpriteBatch.DrawString` method.

The private `RenderGeometryFromVertexBuffer` method receives a `VertexBuffer`, `BasicEffect`, `PrimitiveType` and actually renders the provided geometry with provided shader by calling the `GraphicsDevice.DrawPrimitives` method.

4.2.13. Class Brush

The `Brush` class is an abstract class that is used by all brushes in the library. It defines the `Opacity` property and `Texture2D GetTexture(Size)` method that should be implemented by deriving classes to actually provide their texture representation.

The classes that are included in the library and provide an actual implementation for getting the defining brush texture are the `SolidColorBrush` class, `ImageBrush` class, and the `LinearGradientBrush` class.

4.2.14. Class SolidColorBrush

The `SolidColorBrush` class is responsible for providing a texture colored using a single solid color. For this purposes it exposes public `Color` property.

It provides implementation for the abstract `Brush` class by implementing the abstract `Render` method. During this method a new `RenderTarget2D` instance is created with the size as requested for the brush and set as the active render target. This render target is then cleared by calling the `GraphicsDevice.Clear` method while providing it the requested color. Finally the render target is returned as a result for the `GetTexture` method.

This `GetTexture` method is however usually not used in the case of the `SolidColorBrush` because the GPU can color the rendered vertices with the provided color on its own. This is a more optimal approach then switching active render targets and generating whole texture. Therefore, the usual approach is to get the value of the `Color` property and use it to directly color the individual vertices.

4.2.15. Class GradientBrush

The abstract `GradientBrush` class provides a common ground for all deriving classes that define gradient-based brushes. For this purpose, it exposes a public `GradientStops` property, which is implemented as the `GradientStopCollection` class and defines a collection of `GradientStop` objects.

Class GradientStop

The `GradientStop` class represents a single gradient stop and includes two pieces of information. The first is the `Color` property and the second is the `Offset` property.

Class GradientStopCollection

The class `GradientStopCollection` is a strongly typed collection to be used solely for storing individual instances of the `GradientStop` class. Internally, this class positions its GradientStops elements according to the value of their `Offset` property. The `LinearGradientBrush` relies on this fact when constructing data texture.

Moreover, this class derives from the `Freezable` class, which makes it usable as a *read-only* resource. For this purpose, any public method that can make any changes to the included collection firstly checks internally whether changes are allowed.

Class LinearGradientBrush

The `LinearGradientBrush class` is responsible for providing a texture filled by linear gradient. This gradient is affected by the `GradientStops` property and by two newly defined public properties defined on this class: the `StartPoint` property and `EndPoint` property.

The `LinearGradientBrush` class provides this functionality in its implementation of the `GetTexture` method. As was mentioned, our linear gradient is affected by the `GradientStop` objects that are stored in the `GradientStops` property. Each of these `GradientStop` objects contain two vital information for every gradient stop. The first is the `Color` and the second is the `Offset` of that particular gradient stop. To illustrate this issue, we provide the following Figure 37. On this figure, we can see an example of a linear gradient. The `StartPoint` on coordinates (0,0) and the `EndPoint` on coordinates (1,1) form an interpolation path for the linear gradient. Then, there are four gradient stops, each at different offset and with different color.

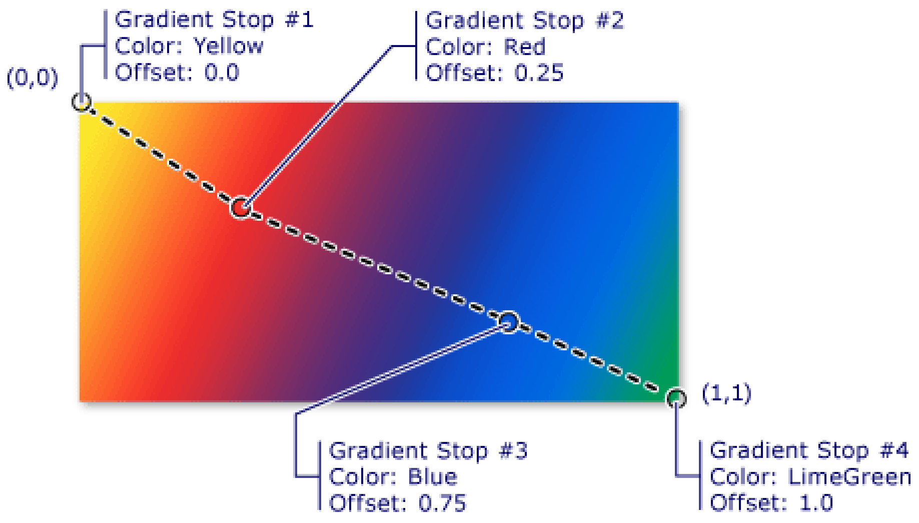


Figure 37: Example of linear gradient with description. Reprinted from [43]

To achieve optimal performance, these linear gradients are created with the help of a graphical shader, coded specifically for this purpose. The functionality of the shader will be described later in this subsection.

Because the number of the gradient stops located in the `GradientStops` collection is varying and the shader cannot be setup to contain a dynamic array to hold all the necessary data, it was decided to create a custom `Texture2D` that will serve as a replacement for this limitation.

The creation of this data texture is the first operation in the `LinearGradientBrush GetTexture` implementation. For the creation of this texture there is private `Texture2D GradientMap` method. In this method a new `Texture2D` is created with the height of two pixels. For the width, the count of the gradient stops is used. Next up, a one-dimensional array of colors is created with the size of `GradientStops.Count * 2`. The array is then filled in a way, in which the color information about the gradient stops are stored first and then, in the same order, the offsets are filled in. Then finally, the `Texture2D.SetData<Color>` method is called to fill the texture with the provided `Color` array. The resulting texture is then in a format where the color information about the gradient stops are located at the zero height index of the texture, while the data about the offsets of these individual stops are stored directly “beneath” them, at the height index of 1.

When this data texture is returned from the `GradientMap` method, a new `RenderTarget2D` is created with the dimensions as required and set as the active render target. At this moment, the linear gradient shader (in the MonoGame represented as an instance of the `Effect` class) is requested from the static class `ResourceDictionary` by calling its `GetResource` method and several values inside the shader are set. These values are `StopCount (int)`, `StartPosition (float)`, `EndPosition (float),` and `GradientStopsMap (Texture2D)`. Now the shader is applied and the render target is rendered into itself. After that, the render target is returned as the desired linear gradient texture.

LinearGradient.fx shader

The `LinearGradient.fx` shader is located at `Content\Shaders\LinearGradient.fx` file in the provided solution for this library. This shader is created and is expected to be compiled with the `ps_4_0_level_9_3` DirectX feature level (that is *Shader Model 3*). It is also necessary to mention that the MonoGame is capable of creating an OpenGL-compatible shaders (GLSL) directly from their DirectX (HLSL) counterparts [44]. Therefore, this shader can be also used for the OpenGL platforms. It is just a matter of substituting the following line at the end of the shader:

```
PixelShader = compile ps_4_0_level_9_3 PS();
```

For the following line:

```
PixelShader = compile ps_3_0 PS();
```

When the shader is set up with the data texture that contains the individual gradient stops and run, the first thing it does is to compute a stepping value for this data texture. This is because the computed data in the texture cannot be accessed directly through pixel coordinates but only in the normalized range from (0,0) through (1,1). This step value is computed by: $1.0\,/\,GradientStopCount$. Finally, by stepping by this amount, the texture sampling would occur directly in-between two pixels, which would distort the read value. That why it is necessary to add another half step whenever the data texture is sampled.

Next up, the offset for the currently processed pixel in relation to the `StartPoint` and `EndPoint` is computed. This is achieved by applying the *Scalar projection* [45] equation.

There are two border case scenarios that are being checked first and in which the final color of the pixel is known immediately. This is when the offset of a given pixel is smaller (or larger) than the lowest (or highest) offset of the gradient stop collection.

This is checked simply, by just sampling the data texture at (0,1) and (1,1) texture coordinates.

If neither of these two cases applies, than it is necessary to go through the individual gradient stop offsets. For this the `for` cycle is used. However due to loop limitations of the *Shader Model 3* based shaders, the number of maximum gradient steps is artificially limited to 12 (set by the `[unroll(12)]` attribute right before the `for` cycle). When it is determined that the offset of a pixel is smaller than the offset of the currently sampled gradient stop, the values of the previous gradient stop and the current gradient stop are stored. Finally it is determined at what offset the pixel is located between the previous and current gradient stop (which produces a value between 0-1), and this information is passed, along with the color of the previous gradient stop and the current gradient stop to the shader `lerp` method. This method produces a linear interpolation of two vectors based and on a weight. The resulting value is then used as a final color for the pixel.

4.2.16. Class TileBrush

The abstract `TileBrush` class further extends the `Brush` class by defining the `Stretch` property. This property determines in the deriving classes how the generated brush should be mapped into the requested brush size.

Class ImageBrush

The only currently provided implementation of the abstract `TileBrush` class is the `ImageBrush` class. This class provides support for using an image in the form of `Texture2D` type as a brush.

This is achieved in the implementation of the abstract `GetTexture` method. Firstly, a new `RenderTarget2D` instance is created with the size as requested for the brush and set as the active render target. Then, it is firstly determined the necessary scale factor for the provided image to reflect the current setting of the `Stretch` property. Finally, a `SpriteBatch` is used to render this image with provided scaling into the active render target. Finally, the render target is returned as a result for the `GetTexture` method.

4.3. Overview of the User input system

Another area of functionality is the *User input system*. The responsibility of this system is to detect changes in the user input and then raise the appropriate events on the individual elements of the user interface. The events are implemented with the concept of *Routed events* in mind with support for both the *Instanced handlers* and the *Class handlers*.

This process can be seen on the following Figure 38. It starts in the moment when the `Update` method of the `PresentationManager` is called. The `PresentationManager` then calls the static `EventManager.RaiseInputEvents`, while providing it with the root element of the user interface, an instance of the `Window` class. Inside this method the `EventManager` checks for inputs that changed (like position of the mouse changed since the last time, a certain keyboard key is pressed now, etc.) and starts the process of raising the appropriate *Routed event*. The methods that start the processing of individual input types are: `RaiseMouseMoveInputEvents`, `RaiseMouseButtonInputEvents`, `RaiseMouseWheelInputEvents`, `RaiseKeyboardInputEvents`. Next step is to get the element where the specific routed event representing the input should be raised. In the case of mouse input, this is done by pure hit testing the *Visual Tree* starting in the root element provided by the `PresentationManager`. In the case of keyboard input we just take the element with keyboard focus (this element is determined during the `RaiseMouseButtonInputEvent` methods). After the hit element is found, all these methods create appropriate variant of `RoutedEventArgs` and fill in the information about the event. Then, an instance of the `EventRoute` class is created, while provided with path for the event and the created routed event arguments. Finally, the `InvokeHandlersOnRoute` method of the `EventRoute` instance is called.

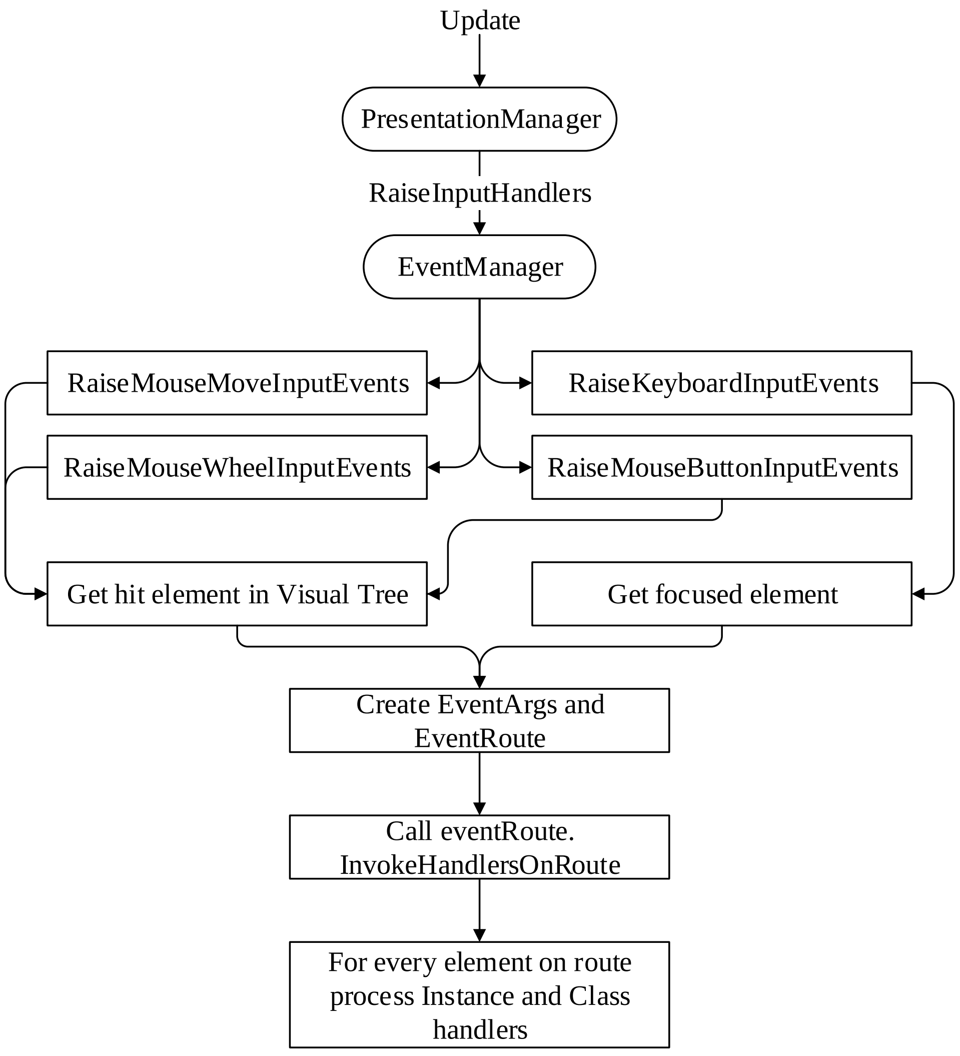
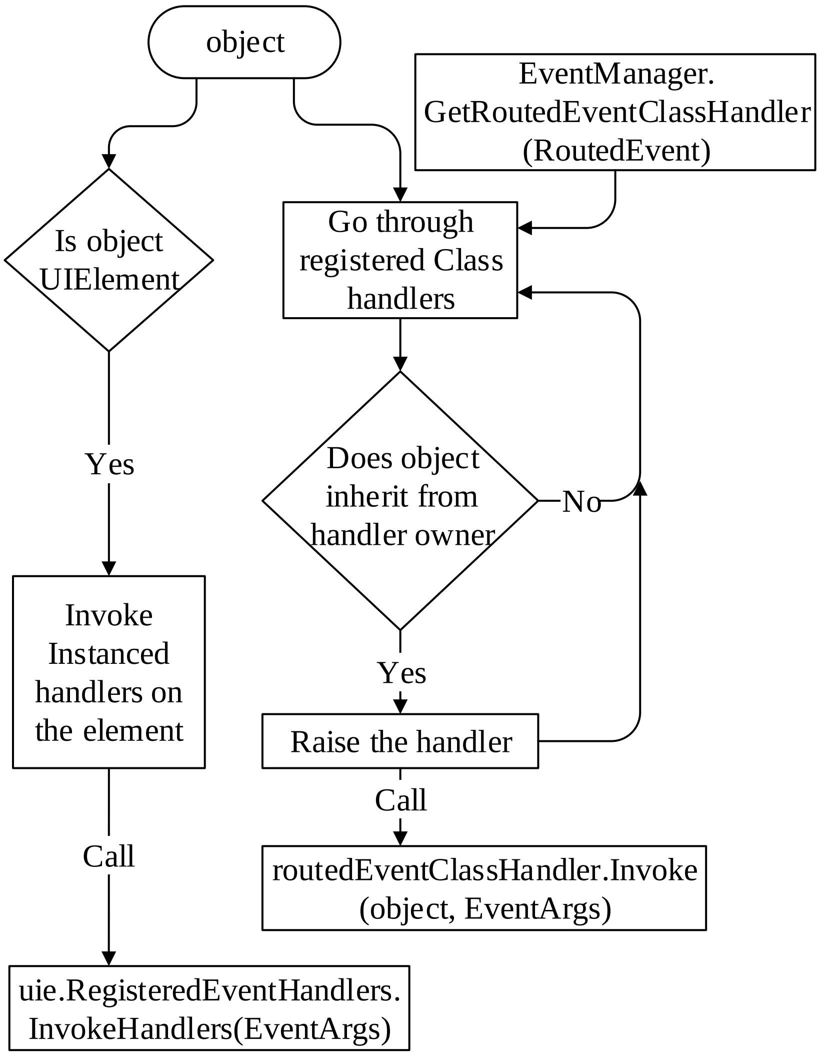

Update
PresentationManager
RaiseInputHandlers
EventManager
RaiseMouseMoveInputEvents
RaiseKeyboardInputEvents
RaiseMouseWheelInputEvents
RaiseMouseButtonInputEvents
Get hit element in Visual Tree
Get focused element
Create EventArgs and EventRoute
Call eventRoute. InvokeHandlersOnRoute
For every element on route process Instance and Class handlers

Figure 38: Process of handling user input #1

This process continues by examining individual elements on the event route. Following process is demonstrated on the Figure 39. During the `InvokeHandlersOnRoute` method `EventRoute` goes through every element (order of elements is determined by the `RoutingStrategy` of the current routed event) and checks for registered *Class* and *Instance handlers* for that element. It firstly gets all the registered class handlers in form of `RoutedEventClassHandler` for a specified routed event by calling the static `List<RoutedEventClassHandler> EventManager.GetRoutedEventClassHandler` method, while providing it the current routed event identification. Then, it goes through this list and for each class handler checks whether the `Type` of currently visited element is equal to or inherits from the owner type of the class handler (specified in `OwnerClass` property of the `RoutedEventClassHandler` class). If this is true, the registered class handler is called by calling its `Invoke` method. Next up, it is determined whether the current element is UIElement. If this is the case, its *Instance handlers* for the current routed event are invoked by the calling the `InvokeHandlers` method of the `EventHandlerStore`, that is located within each `UIElement`.



object
EventManager.
GetRoutedEventClassHandler
(RoutedEvent)
Is object UIElement
Go through registered Class handlers
Does object inherit from handler owner
No
Yes
Yes
Invoke Instanced handlers on the element
Raise the handler
Call
Call
routedEventClassHandler.Invoke
(object, EventArgs)
uie.RegisteredEventHandlers.
InvokeHandlers(EventArgs)

Figure 39: Process of handling user input #2

Raising Class handlers

The individual *Class handlers* are implemented as instances of the `RoutedEventClassHandler` class. When the `Invoke` method is called, it is provided with an `object` and `RoutedEventArgs`. It is firstly checked whether the provided `RoutedEventArgs` are still marked as not handled or whether the current *Class handler* is set to be raised on handled events too. If this check passes, the `Invoke` method checks whether the defined `Delegate` handler (handler that was registered for current *Class handler*) is of type `RoutedEventHandler`, if this is true, then the `Delegate` is called directly by casting it to the `RoutedEventHandler` and provided with the `object` and `RoutedEventArgs`. Otherwise, the `Delegate` is of different type and in this case, the internal `InvokeHandler` method of the provided `RoutedEventArgs` is called.

Raising Instance handlers

The *Instance handlers* are stored in instance of the `EventHandlersStore` class. Every instance of `UIElement` class comes with its own storage. When the `InvokeHandlers` method of an `EventHandlersStore` is called and provided with a `RoutedEventArgs` the `EventHandlersStore` firstly checks its *Instance handler* storage whether there are any handlers registered for the provided routed event (specified in the `RoutedEventArgs`), if so, it goes through all of them and on each it calls its `Invoke` method, while providing it the owner of the storage (a `UIElement`) and the `RoutedEventArgs`. The individual *Instance handlers* are implemented as instances of `RoutedEventInstanceHandler` class. The implementation of the Invoke method is the same like in the case of `RoutedEventHandler`.

Registering routed event

The routed events are represented by the `RoutedEvent` class, whose instance is a unique identifier for a given routed event. A routed event contains several pieces of information. Every routed event has a `string`-based name, defines its RoutingStrategy (*Direct*, *Tunnel*, *Bubble*), `Type` of its handlers, and finally the owner `Type`. The registration of such a routed event is started by calling the static `RoutedEvent EventManager.RegisterRoutedEvent` method. This methods checks whether all the provided information for the new routed event is correct and then creates a new instance of the `RoutedEvent` class while passing it the information provided for this routed event. This new routed event is then registered into the global storage and returned to the caller.

Registering Class handler

The registration process of a *Class handler* begins by calling the static `EventManager.RegisterClassHandler` method. This methods checks whether the provided information about the new *Class handler* is valid. Part of this check is also examination of the provided handler `Type` and the `Type` of the handler as defined for the provided routed event. If they do not match, an exception is thrown. If all checks pass, the new *Class handler* is stored in the global storage for *Class handlers*.

Registering Instance handler

The registration process of an *Instance handler* starts when the `AddHandler` method is called on an instance of the `UIElement`. This method then calls the `AddHandler` method of the internal `EventHandlersStore` and passes it all the received information. If all this information is valid (including the `Type` check on the provided `Delegate`), a new instance of `RoutedEventInstanceHandler` is created and the `EventHandlersStore` inserts it into its storage of local handlers.

4.3.1. Class EventManager

The public static class `EventManager` also serves as the storage for all defined routed events and includes information about all the registered *Class handlers*. The storage for the routed events is defined as internal static `Dictionary<Type, HashSet<RoutedEvent>>` and the *Class handlers* are stored in internal static `Dictionary<RoutedEvent, List<RoutedEventClassHandler>>`.

Besides the previously mentioned methods the `EventManager` also include the public `RoutedEvent[] GetRoutedEvents`, and `RoutedEvent[] GetRoutedEventsForOwner` methods that can be used to access this global storage for routed events.

4.3.2. Class RoutedEvent

The public `RoutedEvent` class serves as unique identification for a routed event and its instance can be created only internally. It includes information about a routed event like its name, `RoutingStrategy`, `Type` of handler, and finally the `Type` of its owner.

4.3.3. Class EventRoute

The public `EventRoute` class is used to invoke all *Class handlers* and *Instance handlers* for given `RoutedEventArgs` on every element on the provided event route. If no route is provided, one is created. For this purpose is used the internal static `GetEventRoute` method that takes an object as a source for the event route and routing strategy according to which the route is created. This method returns a `List<object>` that defines the event route. Last method the `EventRoute` class exposes is the internal `InvokeHandlersOnRoute` method. By calling this method, the `EventRoute` starts invoking the appropriate routed event handlers on the event route.

4.3.4. Class RoutedEventClassHandler

The internal `RoutedEventClassHandler` class represents a single *Class handler*. It contains information about the `Type` of the owner, `Delegate` that should be called and a bool that defines whether this handler should be also invoked for RoutedEventArgs that are already marked as handled.

4.3.5. Class RoutedEventInstanceHandler

The internal RoutedEventInstanceHandler class is very similar to the previously described RoutedEventClassHandler. The only difference is that it does not define the OwnerClass property.

4.3.6. Class EventHandlersStore

The internal EventHandlersStore class provides a storage for *Instance handlers*. This class is being used only with the UIElement and its deriving classes. It provides internal methods to add a new handler (AddHandler) to the handler storage, remove a handler from the storage (RemoveHandler) and finally to invoke registered handlers (InvokeHandlers). The storage for those registered routed event handlers is defined as Dictionary<RoutedEvent, List<RoutedEventInstanceHandler>>.

4.3.7. Class RoutedEventArgs

This public class is a base class for all the other routed event arguments that are used with routed events.

In its base implementation it provides information about the routed event it is associated with, the original source for the raised event and whether the event represented by current instance of RoutedEventArgs was already handled.

This class also defines an infrastructure that is used by the RoutedEventClassHandler and the RoutedEventInstanceHandler when the provided Delegate handler is not of RoutedEventHandler Type. This class defines an internal InvokeHandler(Delegate, object) method that is called in this case. This method then calls the protected virtual InvokeEventHandler(Delegate, object). This method then allows any RoutedEventArgs deriving classes to invoke their appropriate Delegate directly. If not overridden, this virtual method contains a standard implementation that will once again check whether the provided Delegate is RoutedEventHandler, otherwise it will use DynamicInvoke to invoke the specified Delegate.

This library contains several other classes that derive from the RoutedEventArgs class to provide the routed event handlers with additional information about the raised routed events. It is also worth noting that every of these classes is coupled with a specific Delegate handler and every one of them provides an override method for the virtual InvokeEventHandler so the Delegates are called in an optimal way.

Class KeyboardEventArgs

The KeyboardEventArgs provides information about the current keyboard state. It is coupled with the KeyboardEventHandler.

Class KeyEventArgs

The `KeyEventArgs` class further extents the information provided by the `KeyboardEventArgs` class. It introduces information about the keyboard key state changes. This event arguments is used with `KeyUp`, `KeyDown` routed events and their preview versions. This variant is coupled with the `KeyEventHandler`.

Class MouseEventArgs

The `MouseEventArgs` class extends the `RoutedEventArgs` class with the information specific to the mouse input device. It provides information about the state of the mouse left and right buttons and defines a public method `GetPosition` that returns a mouse position to the specified `Visual` instance. This class is used with the `PreviewMouseMove`, `MouseMove`, `MouseEnter` and `MouseLeave` routed events and is coupled with the `MouseEventHandler`.

Class MouseButtonEventArgs

The `MouseButtonEventArgs` class inherits from the `MouseEventArgs` class and provides specific information about mouse button changes and is used with the `MouseDown`, `MouseUp` routed events and their preview versions. It is coupled with the `MouseButtonEventHandler`.

Class MouseWheelEventArgs

The `MouseWheelEventArgs` class extends the information provided by the `MouseEventArgs` class. It provides additional information about the mouse wheel status. It is coupled with the `MouseWheelEventHandler` and is used with the `MouseWheel` and `PreviewMouseWheel` routed events.

4.4. Overview of the property system

The property system of this library is built on the concepts of dependency properties and dependency objects. This concept enforces *thread-safe* access to properties, provides support for resolving the value of a property based on multiple inputs (*value precedencies*) and finally enables the use of *attached properties*.

Like in the WPF, the dependency properties are implemented in the `DependencyProperty` class, the dependency objects in the `DependencyObject` class, and the `Visual` class and all the other classes that participate on the appearance and behavior of the user interface derive from the `DependencyObject` class.

The workings of the property system can be broke down to three stages. These individual stages are, registering a dependency property, setting its value, and finally getting its currently effective value.

Registering a dependency property

Let us firstly look at how a dependency property is registered. This process is illustrated on the following Figure 40. As the `DependencyProperty` serves as a unique identification for a dependency property it is necessary to make sure each dependency property has only one such as identification. This is why there are no public constructors for the `DependencyProperty` class and the only way to create a new `DependencyProperty` instance is to register it through the static `DependencyProperty Register` method (or `RegisterReadOnly` for a *read-only* variant) of the `DependencyProperty` class. Internally the `DependencyProperty` firstly determines whether it was supplied with a `PropertyMetadata` instance for the new dependency property. If this turns to be false, it creates one. After this, the static `int RegisterProperty` method belonging to `DependencyObject` class is called. In this method, the `DependencyObject` determines whether the provided instance of `DependencyProperty` can be registered into the global properties store or not. This is determined by checking whether the properties store already contains a dependency property with the same name that is registered for the same owner. If the property can be registered then the calling `DependencyProperty` class receives back an index of the property in the global store, which is assigned to the property, and the

`DependencyProperty` returns to the caller a new instance of the `DependencyProperty` class.

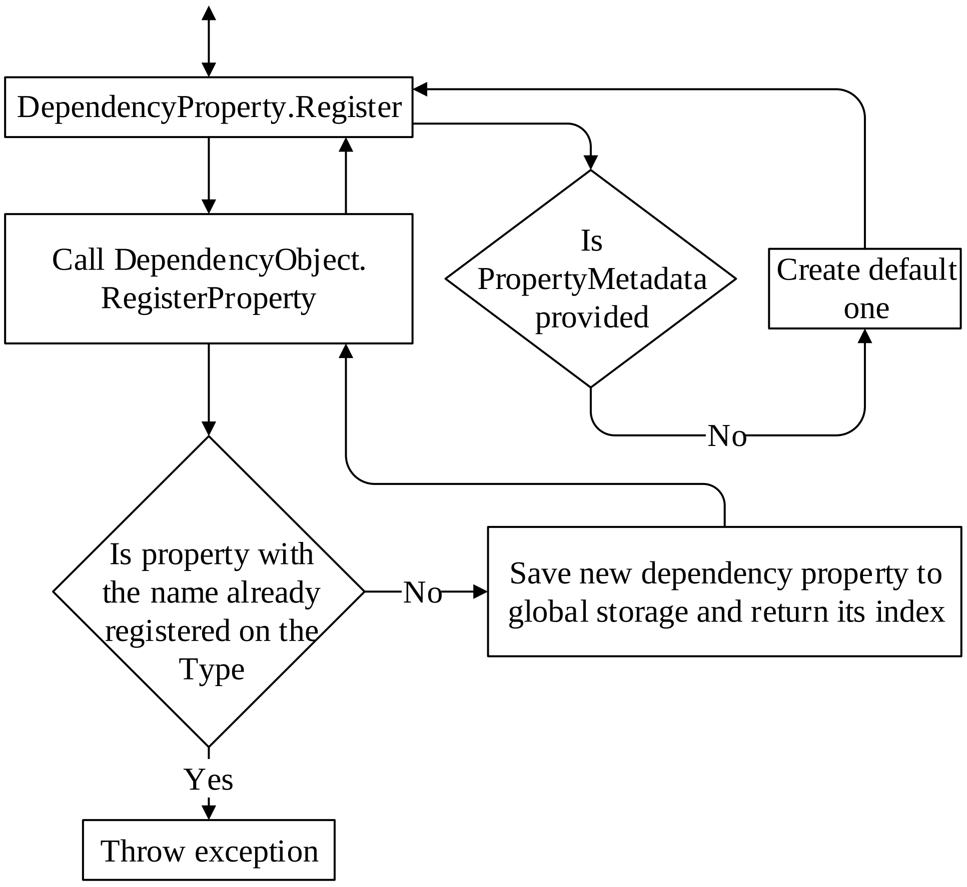

DependencyProperty.Register
Call DependencyObject.
RegisterProperty
Is PropertyMetadata provided
Create default one
No
Is property with the name already registered on the Type
No
Save new dependency property to global storage and return its index
Yes
Throw exception

Figure 40: Process of registering a new dependency property

Setting the value of a dependency property

When the dependency property is successfully registered, it is possible to use its identifier to set its value. The following Figure 41 demonstrates the process of setting a value for a dependency property. The process starts when the `SetValue` method is called on an instance of the `DependencyObject` class where the value is to be set. It firstly checks whether the `SetValue` method was called on the same thread that created the dependency object by calling the `VerifyAccess` method and checks whether the current dependency object is sealed (by checking `IsSealed` property). If any of these two checks fail, an exception is thrown. If the check passes then the system checks if the provided value is correct. This is done in two stages, firstly it is checked whether the `Type` of the provided value is valid for the dependency property and then the system calls a `ValidateVallueCallback` if there is any set for the dependency property. If any of these two checks fails, an exception is thrown.

After the value is validated, the internal `SetValueInternal` method is called. This method allows setting of a value for a dependency property with provided precedence. The SetValue saves values using the Local value precedence, therefore it always passes this value. The `SetValueInternal` checks whether the local storage of values already contains a record for the provided dependency property. If the record does not exist, one is created. This one record for a dependency property is implemented as the `EffectiveValueEntry` class and contains a storage for all the possible precedence values.

Now it is determined whether the effective property metadata for the given dependency property contains a set `CoerceValueCallback`.

If this callback is not defined, the new value is set into the `EffectiveValueEntry` for the given dependency property by calling its bool `SetEffectiveValue` method while providing the precedence value. If the this method returns true, that means the effective value for the property has changed and the dependency object calls the `OnPropertyChanged` method.

Otherwise, if the callback is defined, the system sets the provided value into the `EffectiveValueEntry` with *Local value precedence* but does not listen for value change. Then the `CoerceValueCallback` is called to calculate new value for the property. Once this value is calculated the system once again stores this value into the appropriate `EffectiveValueEntry` by calling its `SetEffectiveValue` method and providing a *Coerce value procedence*. Finally, if this new value changes the effective value for the property, `OnPropertyChanged` method is called.

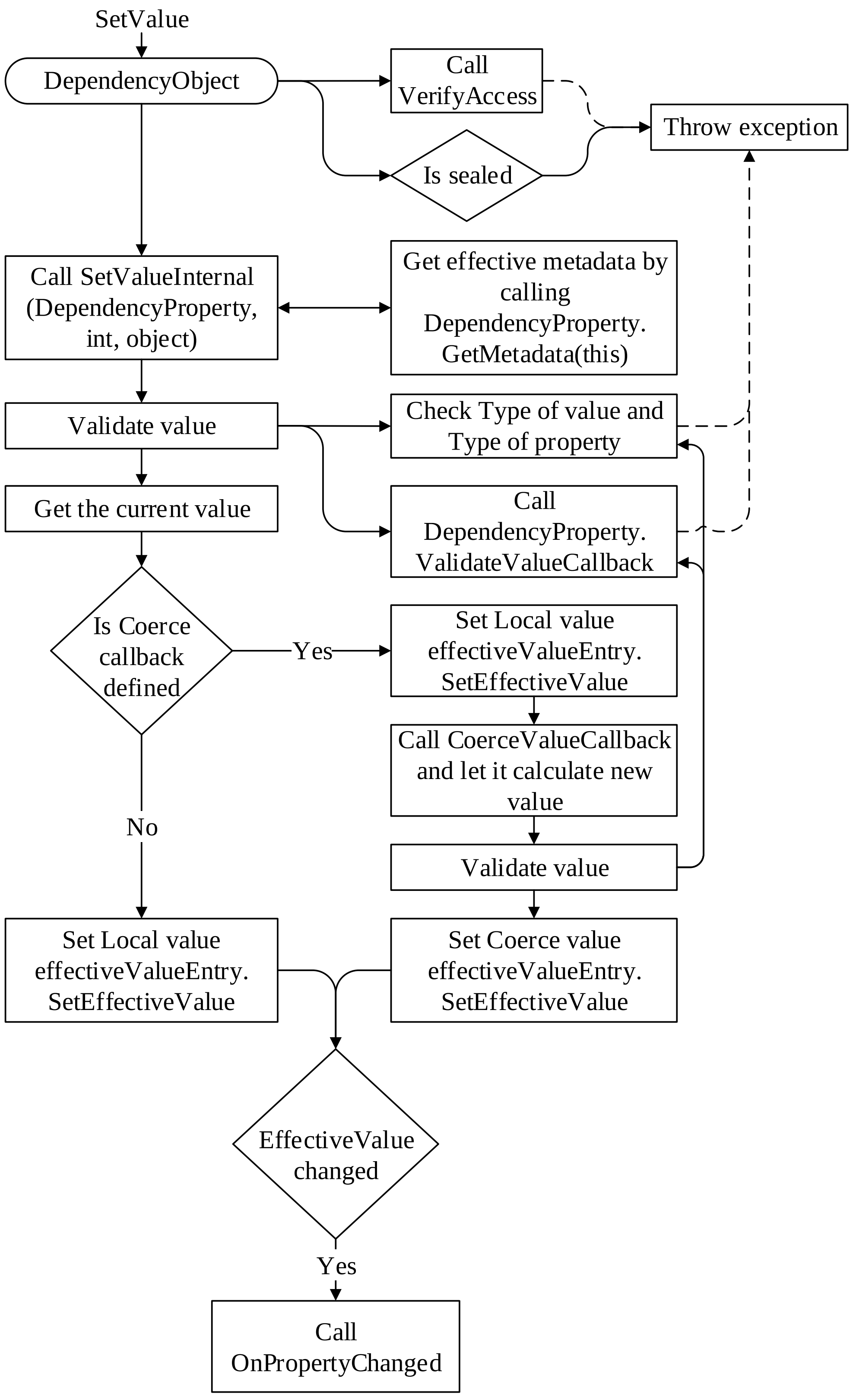

SetValue
DependencyObject
Call VerifyAccess
Is sealed
Throw exception
Call SetValueInternal (DependencyProperty, int, object)
Get effective metadata by calling DependencyProperty. GetMetadata(this)
Validate value
Check Type of value and Type of property
Get the current value
Call DependencyProperty. ValidateValueCallback
Is Coerce callback defined
Yes
Set Local value effectiveValueEntry. SetEffectiveValue
Call CoerceValueCallback and let it calculate new value
No
Validate value
Set Local value effectiveValueEntry. SetEffectiveValue
Set Coerce value effectiveValueEntry. SetEffectiveValue
EffectiveValue changed
Yes
Call OnPropertyChanged

Figure 41: Process of setting a value for a dependency property

Getting the value of a dependency property

The current property system gets the possible value for a dependency property from two possible places. The first place is the local storage of values in every instance of the `DependencyObject` and the second is the default value for a dependency property as given by effective `PropertyMetadata`.

The `GetValue` method returns the currently effective value for a dependency property. The process or getting a value for a given dependency property is demonstrated on the Figure 42. Much like the `SetValue`, the `GetValue` firstly checks whether it can be called from the current thread by calling the `VerifyAccess` method. If this check passes, then it looks up the local value storage for a record for the provided dependency property. If this record exists (once again represented as an instance of `EffectiveValueEntry` class), it asks it to provide the currently effective value by calling its `GetEffectiveValue` method. This value is then returned as a result. On the other hand it the record does not exist the system returns the default value for the given dependency property.

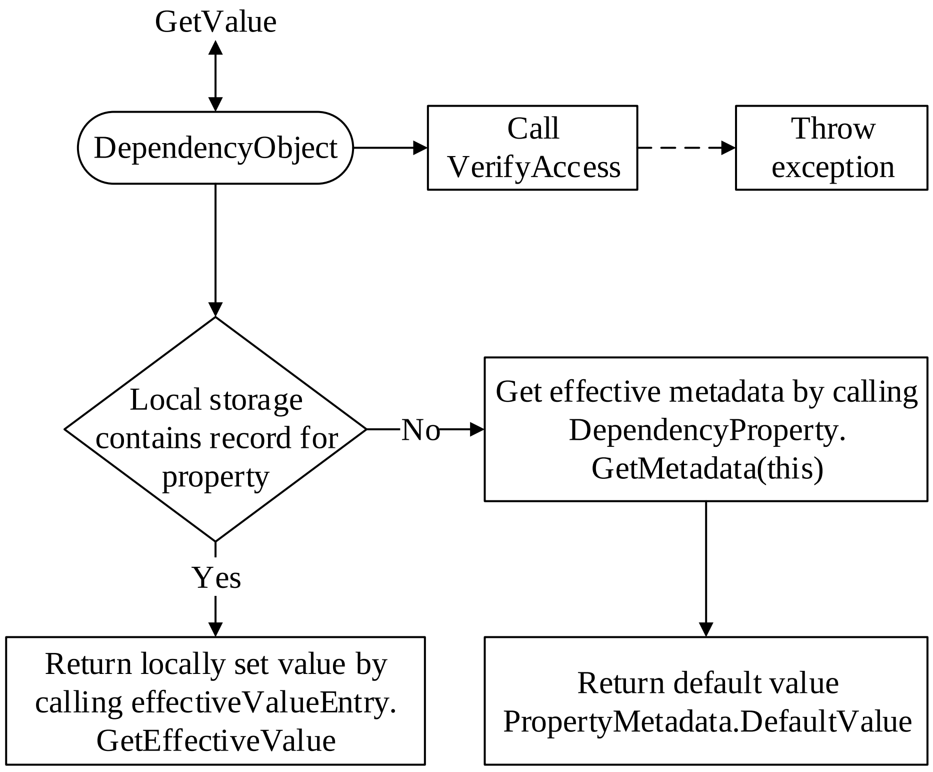

GetValue
DependencyObject
Call VerifyAccess
Throw exception
Local storage contains record for property
No
Yes
Get effective metadata by calling DependencyProperty. GetMetadata(this)
Return locally set value by calling effectiveValueEntry. GetEffectiveValue
Return default value PropertyMetadata.DefaultValue

Figure 42: Process of getting a value for dependency property

4.4.1. Class DependencyProperty

The public `DependencyProperty` class serves as a unique identifier for a certain dependency property. When the static `Register` method is called then the private `RegisterAny` method is called. This method handles all the variants of the `Register` method including `RegisterAttached` and `RegisterReadOnly` methods.

The private GetDefaultPropertyMetadata method determines the default value for the specified property Type and returns a new instance of the PropertyMetadata.

In case of calling the static RegisterReadOnly method, which produces a read-only version of DependencyProperty, the process of creating a new DependencyProperty object has yet another step. After a new instance of the DependencyProperty is created and successfully registered, as described above, a new instance of the DependencyPropertyKey object is created and the DependencyProperty object is passed to it as parameter in the constructor. Once an instance of this DependencyPropertyKey is created, it is returned to the caller and represents a key to the newly created read-only DependencyProperty.

The DependencyProperty instance also allows registering itself to a Type different from the one it was initially registered to. This functionality is exposed through the AddOwner method. The AddOwner method accepts a Type and an instance of PropertyMetadata as its parameters, and internally calls the formerly mentioned DependencyObject.RegisterProperty method, which registers this DependencyProperty to the provided Type. The last step is to register for this Type a PropertyMetadata instance, by calling the OverrideMetadata method while providing it the Type and the PropertyMetadata. The PropertyMetadata instance can be provided by the AddOwner caller, otherwise the default PropertyMetadata of the DependencyProperty is used.

An instance of the DependencyProperty contains a storage for all PropertyMetadata that have been registered for the given DependencyProperty. This storage is implemented as Dictionary<DependencyObjectType, PropertyMetadata>.

As the defined PropertyMetadata is supposed to be applied to the specified Type and all its subclasses until it is overridden by yet another instance of PropertyMetadata, it is necessary to be able to determine the currently effective PropertyMetadata for a given Type. This functionality is implemented in the private GetEffectiveMetadata method. It receives an instance of the DependencyObjectType as a parameter and then it checks the storage. If the dictionary contains the given instance of the DependencyObjectType as a key, it returns the associated PropertyMetadata directly. If the dictionary does not contain it, the BaseType property of the provided DependencyObjectType instance is checked. This provides another instance of the DependencyObjectType that represent its direct Type parent. Now the storage is checked again. This process continues until either the currently checked DependencyObjectType is found in the storage, and in that case, the associated PropertyMetadata are returned as a result, or until the BaseType property of currently checked DependencyObjectType instance is equal to null (the DependencyObject Type itself has been reached). In that case, the DefaultMetadata property of this DependencyProperty is returned.

The GetEffectiveMetadata method is also used to implement the public versions of the GetMetadata methods variants.

The process of setting a custom PropertyMetadata for a Type starts in the public OverrideMetadata method. This method checks whether all the provided parameters are correct and whether the current DependencyProperty is not market as *read-only*. If this is the case and the OverrideMetadata method is not called in its overloaded form that accepts a DependencyPropertyKey, then an InvalidOperationException is thrown. On the other hand, if the key is passed but is not valid for the current DependencyProperty, an ArgumentException is thrown. Then the provided PropertyMetadata and Type are passed to the private OverrideMetadataInternal method. The OverrideMetadataInternal method resolves the provided type into a DependencyObjectType instance by calling the static DependencyObjectType DependencyObjectType.FromSystemType and checks two things. Firstly, it checks whether the local PropertyMetadata storage already contains the resolved DependencyObjectType as a key. If it does, then it means that the default PropertyMetadata have been already overridden for the provided Type and an ArgumentException is thrown. Secondly, it checks whether the default value in the provided PropertyMetadata is valid for this DependencyProperty. If not an ArgumentException is thrown. Finally, if every test passes then the PropertyMetadata is sealed by calling its Seal method and the resolved DependencyObjectType, along with the provided PropertyMetadata are being added to the local PropertyMetadata storage.

4.4.2. Class DependencyPropertyKey

The public DependencyPropertyKey class represents a key that must be used whenever the changes are to be made on a *read-only* DependencyProperty. This includes setting a value for the DependencyProperty on a DependencyObject using the SetValue method or overriding the PropertyMetadata on a DependencyProperty using the OverrideMetadata method. These methods check whether the DependencyProperty property set on the provided DependencyPropertyKey instance matches the DependencyProperty instance where the changes are to be made.

The security of this approach is enforced through internal constructor, which provides the only way to set the included DependencyProperty property.

4.4.3. Class PropertyMetadata

The public PropertyMetadata class allows defining certain aspects of a DependencyProperty. It allows setting a custom default value for the DependencyProperty and defining two callbacks. The PropertyChangedCallback, that is called whenever the value of a DependencyProperty changes and the CoerceValueCallback. The CoerceValueCallback is called whenever the value of a DependencyProperty changes and expects an object to be returned. This returned object is then used by the DependencyObject as the effective value for the DependencyProperty.

This class contains two methods. The first one is the internal `Seal` method. This method is called internally by a `DependencyProperty` instance whenever a `PropertyMetadata` instance is being applied to it. During this method, it is checked whether the provided default value `Type` is `Freezable` (described in more details in section 4.4.9). If this is true, its `Freeze` method is called to make it *read-only*. Finally, the `Seal` method calls the second included method, the virtual `OnApply` method. This last method is not being actively used in the library, but is included because the WPF includes it.

4.4.4. Class FrameworkPropertyMetadata

The public class `FrameworkPropertyMetadata` further extend the information provided by the `PropertyMetadata` about the ways how value changes of the dependency property affect dependency object owner. Individual possibilities are implemented in the `FrameworkPropertyMetadataOptions enum` by using `Flags`.

4.4.5. Class DependencyObject

The public `DependencyObject` class is the base class for the *dependency property system*. An instance of this class can only be created on a thread that contains the `Dispatcher` that is being created as part of the `PresentationManager` initialization process. This fact is checked whenever an instance of the `DependencyObject` created directly in the constructor. If it is determined, that this is not true and the thread calling the constructor is not the same as the thread that owns the `Dispatcher`, an exception is thrown. If the threads match, then a new instance of the `DependencyObject` class is successfully created. This is done so a thread-safe working of the library is assured.

The `DependencyObject` class contains four different storages. The first storage is a storage for all currently registered instances of the `DependencyProperty` class. This storage is implemented by using the `static Dictionary<Type, Dictionary<string, DependencyProperty>>` data structure, where the `Type` represents the individual owners for dependency properties and the nested dictionary provides access to the dependency properties that are registered for a specific `Type` by their names.

Besides this global storage for the registered dependency properties, each instance of the `DependencyObject` also contains a storage for the locally set values of the individual dependency properties. This storage is implemented as `Dictionary<DependencyProperty, EffectiveValueEntry>`. The keys of the dictionary are the individual dependency properties that have their local value set and the values are instances of the `EffectiveValueEntry` class. The `EffectiveValueEntry` class provides the functionality to support the system of value precedencies. The concrete implementation of the `EffectiveValueEntry` class will be covered in the next subsection 4.4.6.

The last two storages are used for the *Binding system* infrastructure. The first storage is implemented as `Dictionary<DependencyProperty, BindingExpression>`

data structure and provides a record of all registered bindings that are currently active on a `DependencyObject`. The second storage is implemented as `Dictionary<string, List<DependencyPropertyChangedEventHandler>>` and allows to set a list of handlers that would be invoked whenever a dependency property with a given name changes. Although this library does not support bindings with complex property paths that define passage through multiple dependency objects, the last `Dictionary` is implemented this way so it could in the future possible support those “bubbling” binding notifications.

The basic functionality of the `DependencyObject` class is to provide a way to read and write values of the dependency properties. This functionality was already described at the section 4.4.

Whenever it is determined that a value for a dependency property changed in the current dependency object, the protected `OnPropertyChanged` method is called. There are three pieces of functionality implemented in this method:

1. Raising `PropertyChangedCallbacks`
2. Calling registered `DependencyPropertyChangedEventHandlers`
3. Processing `FrameworkPropertyMetadataOptions`

The last area of functionality that is implemented in the DependencyObject is support for dependency property bindings. This is achieved by two internal methods.

The first method is the internal `SetBinding(DependencyProperty, BindingExpression, DependencyPropertyChangedEventHandler)`. This method provides a way to register a new binding for a specified dependency property. It firstly determines whether there is already a binding set for the provided dependency property, if this is true, it unregisters it. Then it inserts the provided `BindingExpression` into the local binding storage and provided `DependencyPropertyChangedEventHandler` is registered to the storage of dependency property handlers.

The second method is the `ClearBinding(DependencyProperty)`. This method gets the active binding from the local storage of active bindings and then uses the reference to its `SourceListener` and `TargetListener` handlers to clear them from the storage of dependency property value changed handlers. Finally, it also removes this binding from the storage of active bindings.

4.4.6. Class EffectiveValueEntry

The internal class `EffectiveValueEntry` implements the storage for multiple value precedencies for one dependency property. This storage is implemented in form of `object[]`, where the size of the array is determined by a number of value precedencies (currently ten).

It also provides the public `bool SetEffectiveValue(DependencyProperty, int, object)` that is being used to store specified value with specified value precedence.

The `EffectiveValueEntry` keeps record about the currently highest value precedence and uses this information while determining whether the effective value has changed. In the case when the current highest value precedence is the same as the provided value precedence in the `SetEffectiveValue` method, the current and new object are compared for equality (`object.ReferenceEquals` for reference types, and `object.Equals` otherwise). The `object GetEffectiveValue` is used to return the value with highest currently registered value precedence. While the `object GetEffectiveValue(int)` returns to the caller the value of the requested value precedence record. If the value is not defined, it returns `DependencyObject.UnsetValue` object.

4.4.7. Class DependencyObjectType

The public `DependencyObjectType` class serves as a type cache for all `DependencyObject` deriving types. Such a caching is important because of how the *PropertyMetadata system* with the support for metadata overrides works. This would require a lot of *Reflection* work to determine the effective metadata for a Type.

This class creates a one-way to root referenced tree-like cached structure of relations between various dependency objects. Each instance of the `DependencyObjectType` contains `DependencyObjectType BaseType` property that references the cached representation of the base `Type`, and `Type SystemType` that represents the `Type` of the cached dependency object. The storage for this cached tree is implemented as the static `Dictionary<Type, DependencyObjectType>` data structure.

The public static `DependencyObjectType FromSystemType(Type)` provides a public way of getting the `DependencyObjectType` representation for a given `Type`. This method firstly checks whether the provided `Type` derives from `DependencyObject Type` and then calls internal static `DependencyObjectType FromSystemTypeInternal(Type)`. This method not only returns the appropriate `DependencyObjectType` that represents the provided dependency object deriving `Type` but also caches parts of the tree that are not cached yet.

The following Figure 43 displays how this system works. When the `FromSystemTypeInternal` method is called while provided with a `Type` to get cached representation of, the method firstly checks whether the storage already contains a record for the given `Type`. If it does, it immediately returns the resulting `DependencyObjectType` instance. If the record does not exist yet then the methods calls recursively itself while providing `Type.BaseType Type.` This recursion is repeated until either the appropriate record is found or the cached representation of the `DependencyObject` itself is met. The first element that is manually "cached"-registered to this storage is in the static constructor of this class the `DependencyObject` itself, therefore at the border case every recursion stops there. Finally on the way out of the recursion new instances of every visited `Type` are being created and inserted into the storage.

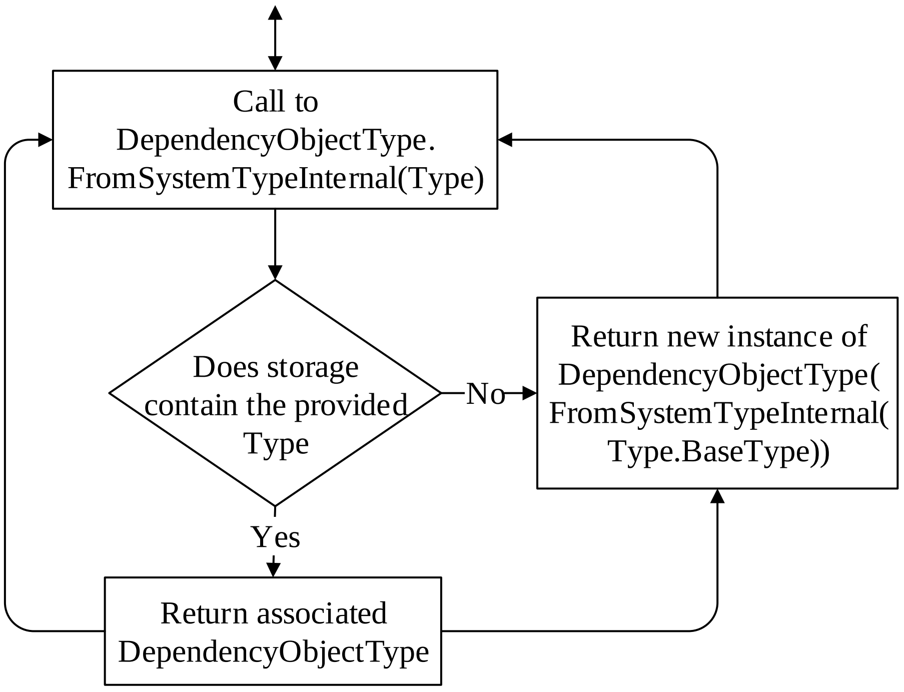

Call to DependencyObjectType. FromSystemTypeInternal(Type)
Does storage contain the provided Type
No
Yes
Return new instance of DependencyObjectType(FromSystemTypeInternal(Type.BaseType))
Return associated DependencyObjectType

Figure 43: Process of caching the DependencyObject types

4.4.8. Class Dispatcher

The `Dispatcher class` provides functionality related to threading that is used heavily in the `DependencyObject` class. It provides public access to the thread the on which the current `Dispatcher` instance was created and a way for a foreign thread to queue operations so they can be executed on the owning thread of the `Dispatcher` instead.

Our library uses only one instance of the `Dispatcher` class, which is why the constructor is marked internal. This one instance of the `Dispatcher` represents the main thread that is used for rendering and processing the user input.

The Dispatcher also contains a queue for all the queued operations that should be executed on its owning thread. This queue is implemented as `Queue<DispatcherOperation>`, where the `DispatcherOperation` represents one queued operation. An instance of the `DispatcherOperation` is created and queued whenever the `Invoke` method is called.

To process all the queued requests the `Dispatcher` exposes the internal `Dispatch` method that goes through every queued `DispatcherOperation` and call its `Invoke` method. This method invokes the queued operation on the thread of the `Dispatcher`.

Class DispatcherOperation

The `DispatcherOperation` represents one operation that is queued to be executed by the `Dispatcher` on its owning thread. This class is used only for internal purposes, and therefore its constructor is marked as internal.

It contains two pieces of information. The first is a `Delegate` that should be invoked by this queued operation and the second is a list of arguments in form of `object[]` that should be passed to the provided `Delegate`. These two things are provided to the `DispatcherOperation` instance in constructor.

Finally, when the `Dispatcher` wants to invoke a stored `DispatcherOperation`, it calls the internal `Invoke` method.

4.4.9. Class Freezable

The public abstract `Freezable` class inherits from the `DependencyObject` class and provides functionality to make itself *read-only*. This is then used for sealing default values of various `PropertyMetadata`.

This functionality is implemented in the public `Freeze` method. The following Figure 44 demonstrates the process of making a `Freezable` object *read-only*. This method sets the private `bool _isFrozen` to `true` (accessible through public `bool IsFrozen` property), so any outside class can check whether the class is writable. Then calls the internal `Seal` method that is defined on the `DependencyObject`, this sets the internal `bool _isSealed` property to `true` and affects the `SetValue` method, which checks this property before it allows any value writes. When the current `Freezable` is sealed, the internal storage for local values (as mentioned in the `DependencyObject` section 4.4.5, this storage is implemented as `Dictionary<DependencyProperty,EffectiveValueEntry>`) is enumerated for values (of *Local precedence*) of dependency properties. This enumeration functionality is implemented in the public `struct LocalValueEnumerator`. This enumerator then goes through the value storage and calls `EffectiveValueEntry.GetEffectiveValue(int)` while asking only for *Local value precedence* values. The `Freeze` method then uses this enumerator to go through all the values. For each value, it checks whether is of type `Freezable`. If this is true, it casts the `object` to `Freezable` and calls its `Freeze` method. This causes a recursive walkthrough of all `Freezable` instances accessible from the initial `Freezable` object. Finally, this method calls the virtual `FreezeCore` method. This method is meant to provide a way to every inheritor that defines any data not stored as dependency properties to seal its stored data.

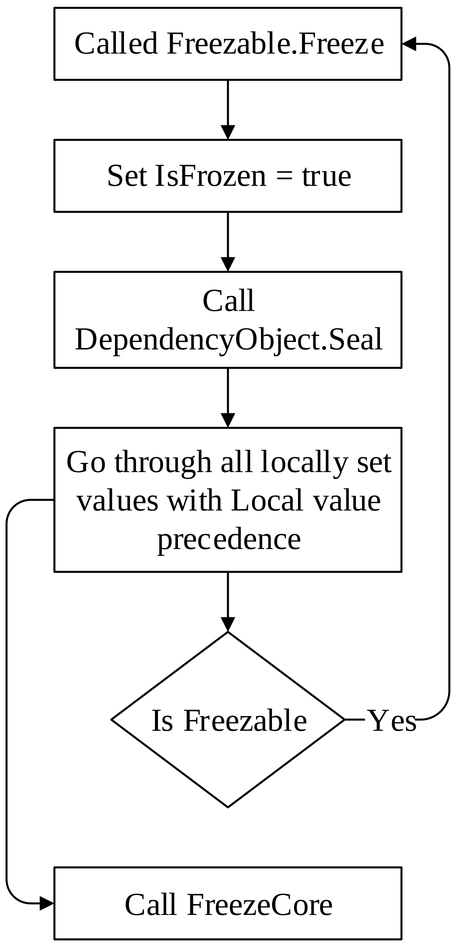

Called Freezable.Freeze
Set IsFrozen = true
Call DependencyObject.Seal
Go through all locally set values with Local value precedence
Is Freezable
Yes
Call FreezeCore

Figure 44: Freezing a Freezable object

An example of usage for this `FreezeCore` in this library is the `GradientStopCollection` class that stores all the `GradientStop` instances in an ordinary `List` that is not stored as `DependencyProperty` nor derives from the `Freezable` class. Therefore, it is necessary to go through every `GradientStop` and `Freeze` it manually.

Finally, the `Freezable` class also implements *Deep cloning* functionality. The process of *Deep cloning* is initialized by calling the public `Freezable Clone` method. The process is demonstrated on the following Figure 45. The `Clone` method firstly calls the protected abstract `Freezable CreateInstance`, which returns a new instance of the given `Freezable`. Then it calls the `CloneCore(Freezable)` method of the newly created object while passing itself as the parameter. Similarly, to the `Freeze` method the values with *Local precedence* are enumerated. However, in this case, this happens on the provided target `Freezable`. Then for every value, it is determined whether it derives from `Freezable` class or not. If not then the target value is set to the current (the new) instance by calling the `SetValue` method. If the value derives from `Freezable`, its `Freezable Clone` method is called and this newly created *Deep copy* of the value is once again saved to the current `Freezable` by calling the `SetValue` method. Once all the values are enumerated, the `CloneCore` methods ends and the `Clone` method returns this newly created and filled `Freezable` object.

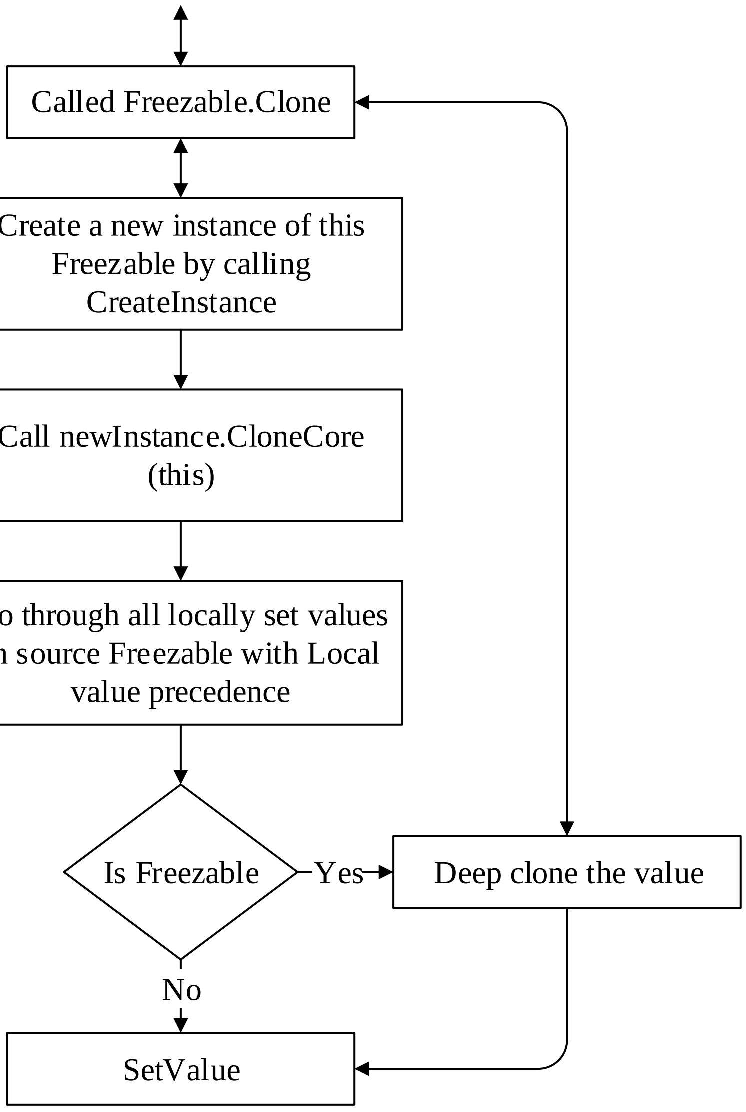

Called Freezable.Clone
Create a new instance of this Freezable by calling CreateInstance
Call newInstance.CloneCore (this)
Go through all locally set values in source Freezable with Local value precedence
Is Freezable
Yes
Deep clone the value
No
SetValue

Figure 45: Deep cloning a Freezable object

4.5. Overview of the binding system

On top of this property system is placed a system for bindings. The binding system implemented in this library provides a way to bind two properties of two objects. Namely, it supports the *OneWay* and *TwoWay* binding modes.

There are always two entities that participate on a binding operation. The first is called the *Source* of the binding, while the second is called the *Target* of the binding. This source object can be any random object. However, the system is designed in a way, that if the source either implements the `INotifyPropertyChanged` or derives from the `DependencyObject` class, the system automatically propagates changes to the *Target*. As for the *Target*, it must always be a `DependencyObject` instance.

The core functionality of the binding system is implemented in the following classes: `Binding`, `BindingExpression`, `BindingOperations`, and `DependencyObject`.

The `Binding` class provides a basic information about the binding, like the source of the binding, and source property that should be bound.

The `BindingExpression` class represents an active binding and provides information about the individual binding participants, status of the binding and also defines two internal `DependencyPropertyChangedHandler`. One is the `SourceListener` and the second is the `TargetListener`.

The functionality to set and remove bindings is implemented in the static `BindingOperations` class.

Finally, an infrastructure to support this binding functionality is implemented in the `DepedencyObject` class itself.

Process of setting up the binding

Let us now look at how the bindings are being set and how they operate. After the static `SetBinding` method of the `BindingOperations` class is called and provided with a `Binding` instance, target dependency object and a dependency property on the target that should be bound, a new instance of the `BindingExpression` is created. This instance is provided with all the information included in the `Binding` instance and the parameters from the `SetBinding` method as well. Then the `Seal` method of the new `BindingExpression` is called. During this method, the system parses information about the source of the binding, determines the source kind (`object`, `INotifyPropertyChanged` object, `DependencyObject`) and tries to locate the source property. If the property is not found on the source object, then the binding sets its `BindingStatus` property to `PathError`. If the source property was resolved successfully, then the binding system tries to set the binding on both the source and target. If the source is `INotifyPropertyChanged` object, then it subscribes to its `PropertyChanged` event. If the source is `DependencyObject`, it calls the internal `SetBinding` method, which registers this binding for the dependency property value change notifications (through registering either the `SourceListener` or `TargetListener` `DependencyPropertyChangedHandler` into list of property value changed listeners). Same process happens for the target `DependencyObject` instance. Finally, the `BindingExpression` calls its `UpdateTarget` method, which gets the value of the bound source property (using *Reflection* for ordinary objects, and using `GetValue` method for dependency objects) and sets it on target (using the public `SetValue` method). Now the process is done and `BindingOperations.SetBinding` return the new instance of `BindingExpression` that represents this new binding back to the caller.

Updating value

Let us now look at how the changes of values are handled.

Firstly, we will describe a model, where there is a binding set up between two `DependencyObjects`, using the *TwoWay* binding. The following process is showed on the Figure 46. Both of these two dependency objects have registered `DependencyPropertyChangedHandler` for the bound dependency property. The source has the `SourceListener` registered, while the target has the `TargetListener`. Whenever a change happens in either of these two dependency objects, the appropriate handler is called. As was already mentioned, these handlers are defined in the `BindingExpression` instance itself and their function is very simple. The `SourceListener` calls the `UpdateTarget` method, while the `TargetListener` calls the `UpdateSource` method. These methods firstly get the current value from the respective source (in this place is also called the `Converter` of the binding, if defined) and then call internally the `SetValue` method on the target dependency object. The `SetValue` checks whether the new value is the same as the one currently set. Therefore, once the value is set on either binding entity, the next one is updated, calls property value changed handlers, and finally the `BindingExpression` tries to update the original entity, it already has the same value set, effectively stopping the value change cascade.

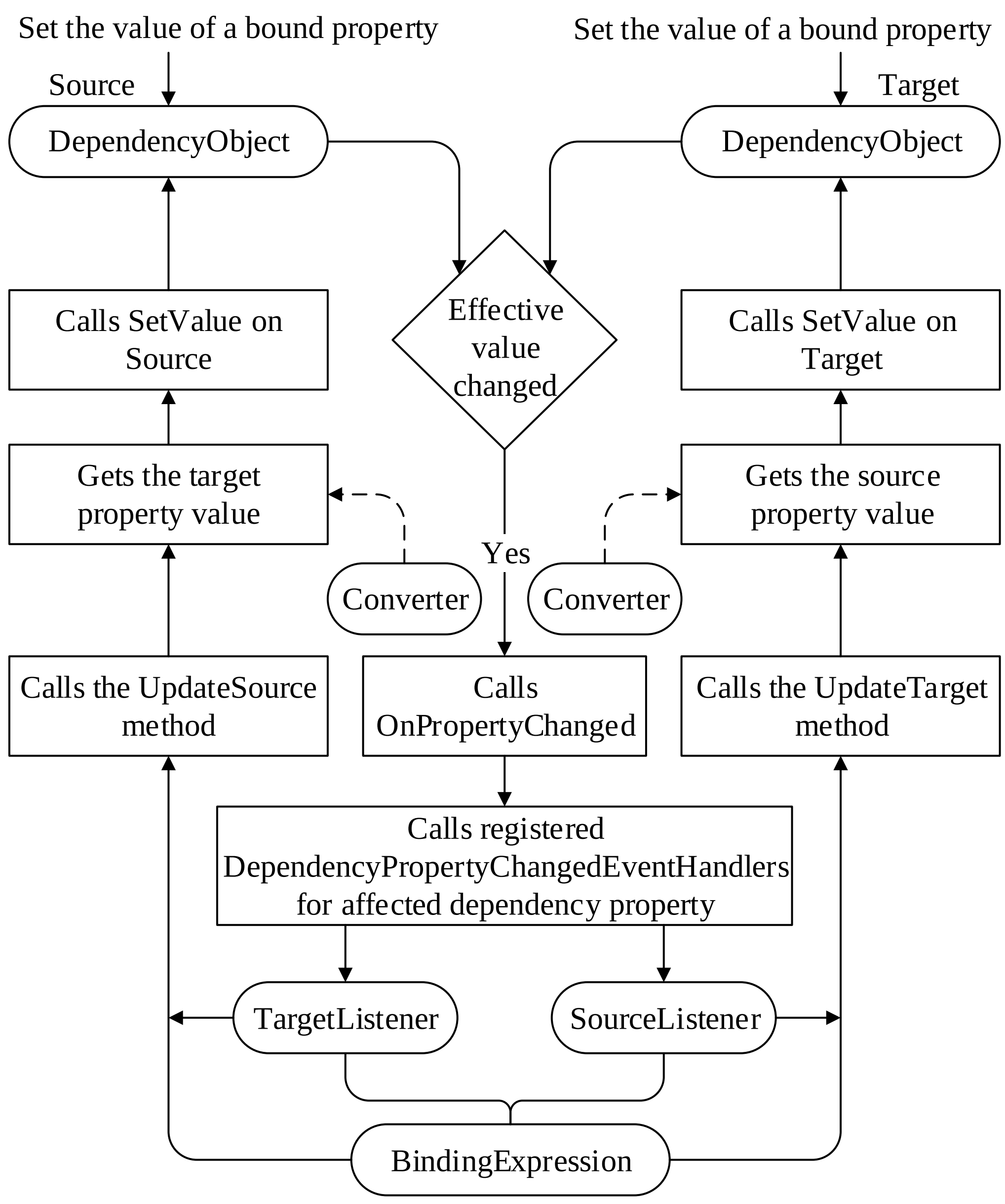

Set the value of a bound property
Set the value of a bound property
Source
Target
DependencyObject
DependencyObject
Effective value changed
Calls SetValue on Source
Calls SetValue on Target
Gets the target property value
Gets the source property value
Yes
Converter
Converter
Calls the UpdateSource method
Calls OnPropertyChanged
Calls the UpdateTarget method
Calls registered DependencyPropertyChangedEventHandlers for affected dependency property
TargetListener
SourceListener
BindingExpression

Figure 46: Process of TwoWay binding between two dependency objects

The situation with *OneWay* is the same, but there is no listening for value changes on the side of the *Target* (no `TargetListener` registered).

The situation for the `INotifyPropertyChanged` object as a source in *TwoWay* binding mode is again similar. The `BindingExpression` is at this case registered on the `PropertyChanged` event, where it checks whether the property that raised the event is the bound source property. If so, it once again calls the `UpdateTarget` method to update the target dependency object. If the value is different from the current one, the dependency object notifies the `BindingExpression` about the change and `BindingExpression` will update the source object using the *Reflection*. This can raise the `PropertyChanged` event on the source once again. However, this value cascade is stopped by the target dependency object that determines that the value is the same as the currently set one and does not raise any further property changed handlers.

4.5.1. Class Binding

The Binding class provides a set of basic information for the biding process. Besides setting the source object for the binding and name of the source property that should be bound, it also allows to define value converter (object that implements the `IValueConverter` interface) that would be used for the binding.

4.5.2. Class BindingOperations

The public static `BindingOperations` provides services in the area of bindings. It also includes a storage that contains references to all currently active bindings. This storage is implemented as `Dictionary<DependencyObject, Dictionary<DependencyProperty, BindingExpression>>`.

4.6. Overview of the control system

This library implements the same structure for the control system as does the WPF. The base class for all controls is the `UIElement` class. This class inherits from the `Visual` class and provides an implementation for its `Render` method. The `UIElement` also include a storage for routed events *Instance handlers* and finally serves as the entry point to the layout system that is also based on the layout system of the WPF and consists of the *Measure* and *Arrange* stages. This layout system is then extended in the `FrameworkElement` class.

4.6.1. Class UIElement

The `UIElement` class contain a storage for routed event *Instanced handlers* (described back in section 4.3.6). Every instance of the `UIElement` comes with own instance of this storage.

The *Measure layout pass* is implemented in the public `Measure` method, which calls internally the protected virtual `MeasureCore` method. The `UIElement` always caches the value provided in the original `Measure` method, so it can be used next time the `Measure` method is called to determine whether the available size has changed or not (if not and the measure is marked as valid, the `MeasureCore` is not being called, thus stopping the recursive `Measure` stage). Moreover, this value is also used by the `LayoutManager` (described in the following section 4.6.2) when the `InvalidateMeasure` method is called on a `UIElement`.

The *Arrange layout pass* is implemented in the public `Arrange` method. This method calls internally the protected virtual `ArrangeCore` method. The `UIElement` always caches the value given in the original `Arrange` method, so it can be used next time the `Arrange` method is called to determine whether the arrange information has changed or not (if not and the arrange is marked as valid, the `ArrangeCore` is not being called, thus stopping the recursive `Arrange` stage). Moreover, this value is also used by the `LayoutManager` when the `InvalidateArrange` (or `InvalidateVisual`) method is called on a `UIElement`.

The `UIElement` allows its inheritors to define their custom rendering instructions. This functionality is exposed in the `OnRender` method through provided instance of the abstract `DrawingContext` class. In the case of this library, this method is called during the `Arrange` method and we pass into this method an instance of the `RenderDataDrawingContext` class (described in the section 4.2.4), which is an internal class that stores rendering instructions for every `Visual` and also implements the abstract `DrawingContext` class.

It also provides several methods to invalidate the layout state of a `UIElement` instance. These methods are the previously mentioned `InvalidateMeasure`, `InvalidateArrange`, and `InvalidateVisual`. When called, these methods mark internally the appropriate layout pass as invalid and queue this element for a new layout pass using the static `LayoutManager` class.

Finally, the `UIElement` class provides an implementation for the abstract `Visual.Render(RenderContext)` method. During this method, `UIElement` adds its `VisualOffset`, `VisualOpacity`, and `VisualClip` to the `RenderContext.` Then it renders its own rendering instructions, and finally calls the `Render` methods on all its `Visual` children. When children finish with their own rendering the `Render` method starts the process of “clean up” and removes its `VisualClip`, `VisualOpacity`, and finally the `VisualOffset` from the `RenderContext`.

4.6.2. Class LayoutManager

The internal static `LayoutManager` class serves as a storage for all the layout requests coming from `UIElement` objects and when asked by the `PresentationManager` it also processes them.

These layout requests are the *Measure* and *Arrange* requests. A request is queued when either the `InvalidateMeasure` or the `InvalidateArrange` method is called (also indirectly by `InvalidateVisual`, as this method calls the `InvalidateArrange` internally) on any instance of `UIElement`. The requests are stored in two private queues of type `Queue<UIElement>,` where one is dedicated to *Measure* requests and the second one for *Arrange* requests.

A `UIElement` can is queued for a specific request by using one of the following static methods: `EnqueueMeasureRequest`, `EnqueueArrangeRequest`. Each of these two methods accept a `UIElement` object as a parameter and queues it to the right queue.

The entry point for processing all the queued requests is the public static `ProcessRequests` method. When this method is called, the *Measure* and *Arrange* queues are processed and the `LayoutManager` goes through both queues, all queued elements, and with every UIElement object calls either its Measure method or the Arrange method (depending on which queue it is processing). As parameters for these two methods, the LayoutManager uses the cached values that were used last time when the Measure and Arrange methods were called on the UIElement. These cached values are defined as internal and stored directly on the individual UIElement objects.

Finally, the LayoutManager also includes information indicating whether the user interface should be redrawn. This information is exposed in the public static bool ShouldRedraw property and is manipulated using two methods. The public static InvalidateVisualTree method sets this property to true, while the Redrawn method sets it to false. The firstly mentioned method is being called as a part of the InvalidateVisual method on a UIElement instance, and also whenever VisualClip property of a Visual instance changes. The Redrawn method is being called by the PresentationManager to notify the LayoutManager about the fact that the user interface was successfully redrawn.

5 Library API documentation

In this chapter, we will look at the user side of this library, therefore the programmer. We will go through the requirements to use this library and show how to setup a MonoGame project for use with this UI library and provide a few code examples. Finally, we will go through the list of implemented features.

5.1. Requirements

To use this library in a project, it is necessary to have the .NET Framework 4 (or newer) and the MonoGame 3.4 (or newer) installed. The installation file for this version of the MonoGame is included as the **Attachment C**. Moreover, the compiled version of this library as provided across the attachments is compiled to be used with the `MonoGame Windows Project`. The `MonoGame Windows OpenGL Project` is not currently supported.

On the hardware side, this library requires a GPU with support for at least the Shader Model 3.

5.2. How to set up a project

There are two possible ways how to set up a project. First way is to simply use the provided template (**Attachment D**, only for *Visual Studio*) and everything gets set up automatically. Alternatively, the project can be set up manually using the following steps.

Start by creating a new `MonoGame Windows Project`. Then reference the provided `MonoGameWPF.dll` to the project and add the provided `LinearGradient.xnb` file into a folder named `MonoGameWPF` inside the root of your `Content` folder. This is the place where the library will be looking for the content file. This managed assembly and a content file, contain all the necessary functionality of this library.

Now open the `Game1.cs` file that was created along the new MonoGame project and declare somewhere in the code of this class a variable that will contain the instance of `PresentationManager`. This instance will be accessed along the entire life span of the app.

```
public partial class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    …
    // Define a variable for PresentationManager
    PresentationManager presentationManager;
    //
    …
}
```

Figure 47: Defining variable for PresentationManager

In the `Initialize` method of the `Game1` class, create a new instance of the `PresentationManager` and save it to the previously defined variable.

```
protected override void Initialize()
{
    // Initialize a new instance of PresentationManager
    // Pass it the reference to your Game class, GraphicsDeviceManager,
    // and finally specify whether to enable window resizing
    presentationManager = new PresentationManager(this, graphics, true);

    base.Initialize();
}
```

Figure 48: Creating a new instance of PresentationManager

In the `LoadContent` method of the `Game1` class, call the `LoadContent` method of the newly created `PresentationManager` instance. During this method, the `PresentationManager` loads up the `LinearGradient.xnb` content file.

```
protected override void LoadContent()
{
    // Call the LoadResource method and pass it instance of ContentManager
    presentationManager.LoadResource(Content);
}
```

Figure 49: Loading content

Finally, the `PresentationManager` needs to be able to react on input changes and to actually render its content. Therefore, a call to the `Update` method of the `PresentationManager` needs to be placed into the `Update` method of the `Game1` class and the calls to the `PreRender` and `Render` methods (in this order) of the `PresentationManager` inside the `Draw` method of the `Game1` class.

```
protected override void Update(GameTime gameTime)
{
    // Raise input events and process Dispatcher
    presentationManager.Update();
}
```

Figure 50: Listening for updates

```
protected override void Draw(GameTime gameTime)
{
    // Prepare the visual representation of the user interface
    presentationManager.PreRender(gameTime);
    // Clear the screen
    GraphicsDevice.Clear(Microsoft.Xna.Framework.Color.White);
    // TODO: Add your game drawing code here

    // Draw the user interface onto the screen
    presentationManager.RenderUI();
}
```

Figure 51: Structure of the Draw method

This concludes the setting process for this library. Then, any custom game drawing instructions must be placed between the `PreRender` and `Render` methods.

5.3. How to load up fonts

This library uses the standard MonoGame font system based on the `SpriteFont` class.

By default all controls of this library look for a font family named “DefaultFont”. This font is being loaded up from a file named `DefaultFont.xnb` located in the same place as the `LinearGradient.xnb` (folder named `MonoGameWPF` inside the root of your `Content` folder). This font is registered during the content loading stage of the `PresentationManager` (as seen in section 5.2, Figure 49) so the controls can use it.

There are two fonts included with this library for every developer to use right away. Those fonts files are located in the **Attachment E** and are called `DefaultFont.xnb` and `DefaultFontItalic.xnb`. They are based on the *Ubuntu-M* and *Ubuntu-RI* fonts and have been downloaded from the Ubuntu website [46].

Now we will go through the process of registering any additional fonts to be used with this library.

This process consists of two steps. Firstly, the font needs to be loaded using the MonoGame `ContentManager`. And secondly this new `SpriteFont` must be registered as a font resource for this library. This achieved by calling the `ResourcesDictionary.RegisterFont` static method, while providing it the loaded `SpriteFont` and the desired font family name. The following two lines of code on the Figure 52 illustrate the process of registering a new font:

```
// Standard way of loading Fonts in MonoGame
SpriteFont NewFont = Content.Load<SpriteFont>(PathToFontXNBFile);
// Now register this loaded font as a resource for this library
ResourcesDictionary.RegisterFont(NewFont, FontFamilyName);
```

Figure 52: Registering a font

5.4. Examples

In this sub-chapter, we will provide some examples of how to work with this library and what can be achieved. We will also expect that the appropriate solution for use with this library (as described in the section 5.2) has been already created.

Finally, all the following examples will be defined in the `LoadContent` method of the `Game` class, unless specified otherwise. In addition, if the project was created manually it might be necessary to add the following namespaces:

- `System.Windows`
- `System.Windows.Controls`

- `System.Windows.Data`
- `System.Windows.Input`
- `System.Windows.Media`
- `System.Windows.Shapes`

All the following examples are included in the form of individual Visual Studio solution in the **Attachment F**.

5.4.1. Hello world

We will start with a simple example. We are going to put a `TextBlock` control into our custom user interface and make it to display a *"Hello world"* text.

The code for this example can be seen on Figure 53. Firstly, start by creating a new instance of the `TextBlock` class. This control is designed specifically to be used for displaying text. Then, use its `Text` property and set it to the "Hello world" text.

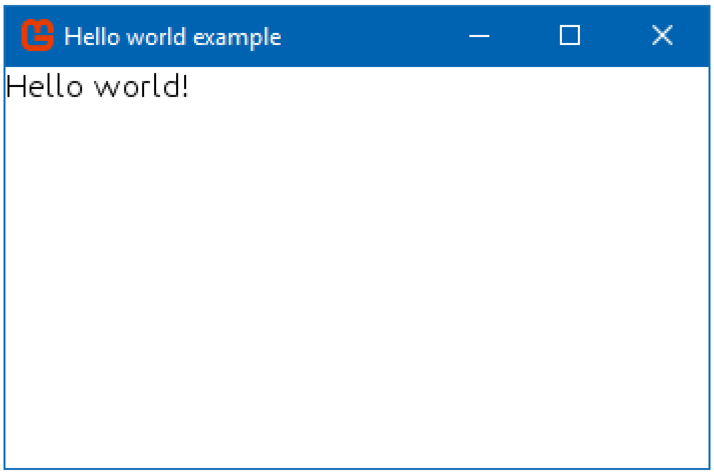
At this point, this control exists and has a `Text` property set to out custom text. However, this control is still not connected in any way to the `PresentationManager` therefore, it will not be rendered. To connect this newly created `TextBlock` we are going to use the `Window` property of the `PresentationManager`. This property serves as an entry point to adding custom controls into the UI.

Next up, we set our `TextBlock` instance as the value for the `Content` property of the mentioned `Window`. This connects the `TextBlock` to the root of the user interface and allows it to be rendered.

```
protected override void LoadContent()
{
    // Part of the PresentationManager initialization process
    presentationManager.LoadResource(this.Content);

    // Creates new instance of the TextBlock
    TextBlock textBlock = new TextBlock();
    // Sets the "Hello world!" as its text
    textBlock.Text = "Hello world!";

    // Assigns the textBlock into the user interface tree
    presentationManager.Window.Content = textBlock;

}
```

Figure 53: Hello world code example

When run, this code produces what can be seen on the following Figure 54:



Hello world!

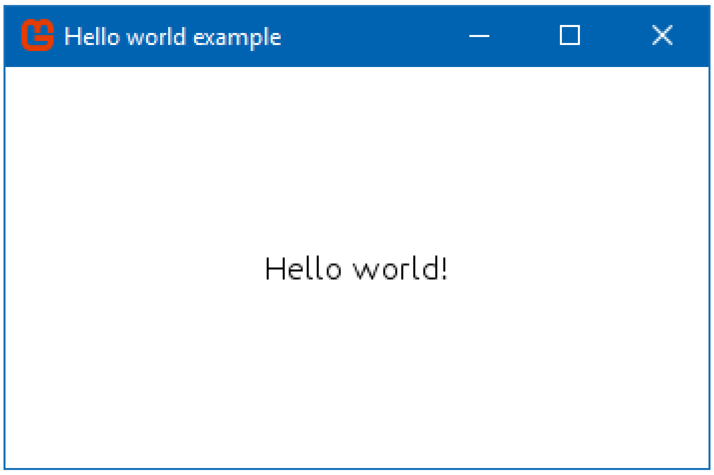
Figure 54: Hello world example screen #1

However, we would like the text to be centered in the middle of the window. Because the `TextBlock` control derives from the `FrameworkElement`, we can use its `HorizontalAllignment` and `VerticalAllignment` properties to position it accordingly. Let us add the following lines of code to the previously written code:

```
textBlock.HorizontalAlignment = HorizontalAlignment.Center;
textBlock.VerticalAlignment = VerticalAlignment.Center;
```

Figure 55: Setting alignment properties

Now when the above code is added, let us run the application again. On the following Figure 56 we can see the result:



Hello world!

Figure 56: Hello world example screen #2

We can see the `TextBlock` control, showing us the provided text while being perfectly aligned to the middle of the window, as requested.

5.4.2. Button example

Now we will showcase something more complex. In this example, we will use the `Button` control and its `Content` property to create a customized look. For this new look we will use the `Ellipse` and `TextBox` controls.

We start by creating a new instance of the `Button` control. We would also like the control to be in the middle of the screen like in the last example. For this, we reuse the code from the Figure 55. Then we need to create instances of the `Ellipse` control and the `TextBlock` control.

However, for the `Ellipse` to actually display, it is necessary to set its `Width` and `Height` properties along with the brush of its fill. We are going to use some smaller number for the size example, like 50 pixels. As for the fill, there are three possible types of brushes that can be used in this library: solid color, linear gradient, and image. For this example will use simple green color. On the following Figure 57 we can see our code so far:

```
protected override void LoadContent()
{
    // Part of the PresentationManager initialization process
    presentationManager.LoadResource(this.Content);

    // Create a new instance of the Button control
    Button button = new Button();
    // Set its alignment properties, so it stays in the middle of the
    // window
    button.HorizontalAlignment = HorizontalAlignment.Center;
    button.VerticalAlignment = VerticalAlignment.Center;

    // Create a new instance of the Ellipse control
    Ellipse ellipse = new Ellipse();
    // Set its size
    ellipse.Width = 50;
    ellipse.Height = 50;
    // Set the fill of the ellipse to green color
    ellipse.Fill = new SolidColorBrush(Color.Green);

    // Create a new instance of the TextBlock control
    TextBlock textBlock = new TextBlock();
    // Set its text
    textBlock.Text = "Hello world";
    // Assigns the button into the user interface tree
    presentationManager.Window.Content = button;

}
```

Figure 57: Button example code #1

On the previous Figure 57 can be seen the first difference between our library and the WPF. The WPF uses the `Colors` class to store all the predefined colors, while this library uses the stock MonoGame color definitions, which are stored at the `Color` class.

The last problem we are facing is how to set both the `TextBlock` and the `Ellipse` as the `Content` of the `Button` control. The `Content` property of the `Button` class accepts a single `object`, therefore it is necessary to store our two controls in some container control. We will position our controls one above the other and for this is the most suitable the `StackPanel` control. After we create a new instance of the `StackPanel` control, we need to add our controls as the children for this container, for this, we use its `Children` property, which exposes the `Add` method. Into this method, we gradually pass our two controls. Finally, we add the `StackPanel` as the `Content` for the `Button`. On the following Figure 58 is the code we added since the beginning of this paragraph:

```
// Creates a new instance of the StackPanel container control
StackPanel panel = new StackPanel();
// Now add both our controls
panel.Children.Add(ellipse);
panel.Children.Add(textBlock);
// Set the StackPanel as the Content of the Button
button.Content = panel;
```

Figure 58: Button example code #2

Now, if we were to run this code, we would get the result as can be seen on the following Figure 59:

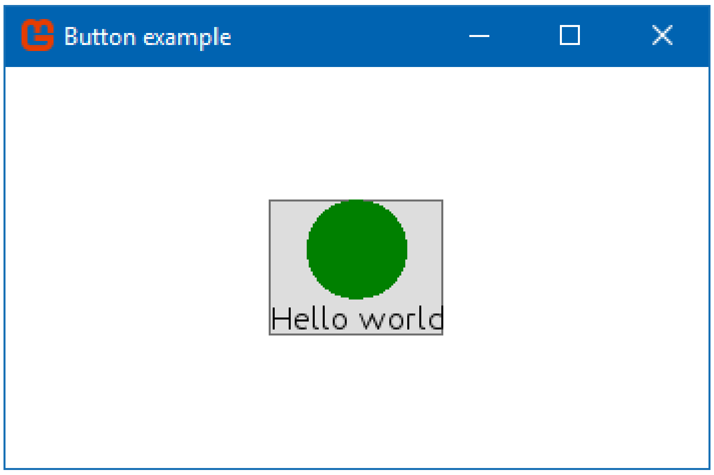


Figure 59: Result of the previous Button example code

However, this is the `Button` control. It is used to perform some action when the user clicks on it. For this purpose, the `Button` defines the `Click` event. We are going to demonstrate this event by changing the color of our `Ellipse` from green to yellow. For the purpose of easier demonstration, we are going to register for this event using a *Lambda Expression*. Therefore, the following code was added to our previous code:

```
button.Click += (s, e) =>    {
    ellipse.Fill = new SolidColorBrush(Color.Yellow);
};
```

Now, if we run again our code example and click on the `Button` control, the included `Ellipse` will turn yellow. The final result of this example can be seen on the following Figure 60:

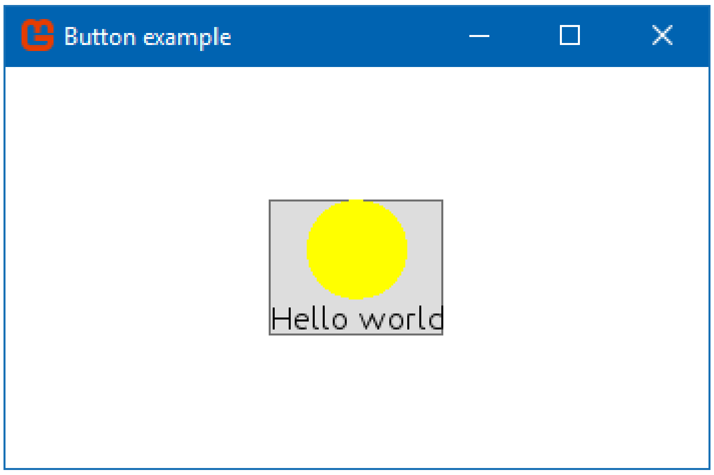


Figure 60: Result of the Button example code after clicking on the Button

5.4.3. Fonts example

For this example we are going to showcase loading another font to use with this library and using two instances of the `TextBlock` control to display text using different fonts.

We will start by putting our custom MonoGame font file into the root of the `Content` folder. For this example we are going to use the included `DefaultFontItalic.xnb`.

Like in the previous two examples, we write our code as part of the `Game1.LoadContent` method. On the Figure 61 we can see the code for first part of this example. Firstly, the provided font needs to be loaded into the MonoGame and saved as a `SpriteFont` instance using the standard MonoGame `ContentManager` whose instance is available as part of the `Game1` class. Secondly, this `SpriteFont` must be registered into the resource dictionary of this library and provided with a font family name. We are going to name this new font as "Italic font".

```
// Load the custom font into the MonoGame
SpriteFont DefaultFontItalic =
    Content.Load<SpriteFont>("MonoGameWPF\\DefaultFontItalic");
// Register the font with the library
ResourcesDictionary.RegisterFont(DefaultFontItalic, "Italic font");
```

Figure 61: Loading a custom font

The code for both the first part of this example and the following second part can be seen on the Figure 62 below. Now we create two instances of the `TextBlock` control and set them both with yellow background and centered horizontal alignment. Moreover, we set the first `TextBlock` control to show “Hello!” text, while the second one we set up to show “Hello Italic!” text. Finally, we create a new instance of `FontFamily` class, provide it with the name of our custom font, “Italic font”, and set it as `FontFamily` property of the second `TextBlock`. In the end, to put both of these `TextBlock` controls onto the screen we need to use a container control. We are going to use again the `StackPanel` control and also set it to be centered on the screen.

```
protected override void LoadContent()
{
    // Part of the PresentationManager initialization process
    presentationManager.LoadResource(this.Content);

    // Load the custom font into the MonoGame
    SpriteFont DefaultFontItalic =
        Content.Load<SpriteFont>("DefaultFontItalic");
    // Register the font with the library
    ResourcesDictionary.RegisterFont(DefaultFontItalic, "Italic font");

    // Prepare a new black color brush
    SolidColorBrush yellowColor = new SolidColorBrush(Color.Yellow);

    // Initialize first TextBlock
    TextBlock textBlock = new TextBlock();
    textBlock.Text = "Hello!";
    textBlock.Background = yellowColor;
    textBlock.HorizontalAlignment = HorizontalAlignment.Center;

    // Initialize second TextBlock
    TextBlock textBlock2 = new TextBlock();
    // Here create a new instance of FontFamily
    // and provide it with the name of our custom font.
    textBlock2.FontFamily = new FontFamily("Italic font");
    textBlock2.Text = "Hello Italic!";
    textBlock2.Background = yellowColor;
    textBlock2.HorizontalAlignment = HorizontalAlignment.Center;

    // Create new StackPanel to hold our TextBlock controls
    StackPanel panel = new StackPanel();
    // Set it to remain in the center of the screen
    panel.HorizontalAlignment = HorizontalAlignment.Center;
    panel.VerticalAlignment = VerticalAlignment.Center;

    // Add these two TextBlock controls to our StackPanel
    panel.Children.Add(textBlock);
    panel.Children.Add(textBlock2);

    presentationManager.Window.Content = panel;
}
```

Figure 62: Font example code

If we were to compile and run the following code, we would get the user interface as seen on the following Figure 63:

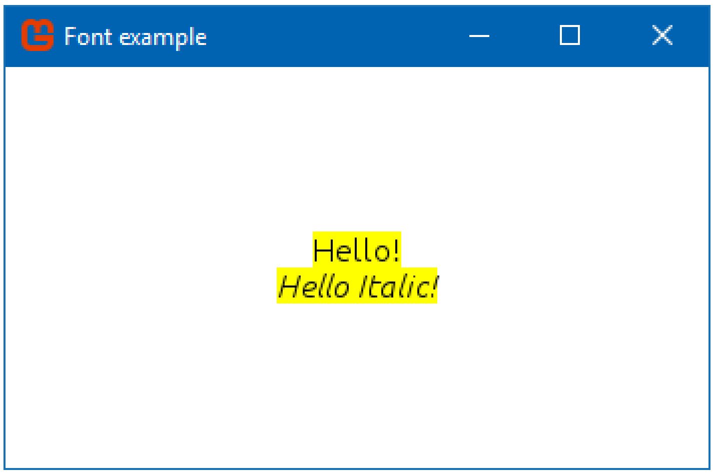


Figure 63: The result of font example

5.4.4. Data binding example

In this example we are going to show how to create a *Data binding* between two dependency objects and two dependency properties. To showcase this, we are going to use the `TextBox` control and the `TextBlock` control. We are going to set up the data binding in a way where whenever a text is inputted to the `TextBox`, the `TextBlock` is updated with this text as well.

We will start by creating an instance of the `TextBox` and `TextBlock` controls. We set the `TextBox` control to be located on the bottom-left corner of the window, while the `TextBlock` will be located in the bottom-right corner. We will also set the background of the `TextBlock` control to `SlateGray` color, so we can see the area taken by it. On the following Figure 64 we can see our code so far:

```
// Create a new TextBox and place it in bottom-left corner
TextBox textBox = new TextBox();
textBox.Width = 200;
textBox.Height = 50;
textBox.HorizontalAlignment = HorizontalAlignment.Left;
textBox.VerticalAlignment = VerticalAlignment.Bottom;
textBox.VerticalContentAlignment = VerticalAlignment.Center;

// Create a new TextBlock and place it in bottom-right corner
// This TextBlock will be always target for the binding
TextBlock textBlock = new TextBlock();
textBlock.HorizontalAlignment = HorizontalAlignment.Right;
textBlock.VerticalAlignment = VerticalAlignment.Bottom;
// Set background so we can see the area of the TextBlock
textBlock.Background = new SolidColorBrush(Color.SlateGray);
```

Figure 64: Binding example, setting up controls

As we are once again using more than one control, we need to use some container control. In this case we are going to use the `Grid` control and place these two controls inside it.

Finally, we are going to set up our data binding. This process consists of two parts. In the first part, we create a new instance of the `Binding` class. The `Binding` class allows us to define the source object of the binding, name of property on the source that should be bound, and finally the binding mode. For this example we are going to set our `TextBlock` instance as the source object of the binding and its `Text` property as the source property. As for the binding mode, we are going to set the `OneWay` mode. We can see the code for creating a new `Binding` and setting its properties on the following Figure 65:

```
// Create a new Binding
Binding binding = new Binding();
// Set the TextBox as our source object
binding.Source = textBox;
// Set the source property as the Text property
// (this is a dependency property)
binding.Path = "Text";
// Set OneWay binding mode
binding.Mode = BindingMode.OneWay;
```

Figure 65: Setting up a Binding object

Secondly, we need to apply this binding definition to a target and its dependency property. This is done through the static `BindingOperations` class, by calling its `SetBinding` method, as illustrated on the following Figure 66:

```
BindingExpression bindingExpression =
    BindingOperations.SetBinding(
        textBlock,              // Target dependency object
        TextBlock.TextProperty, // Target dependency property
        binding);               // Binding definition
```

Figure 66: Applying binding to a dependency object

Finally, if we run this code and write something to the bound `TextBox` control then this text would be immediately transferred to the `TextBlock` control. We can see the result of this example on the following Figure 67:

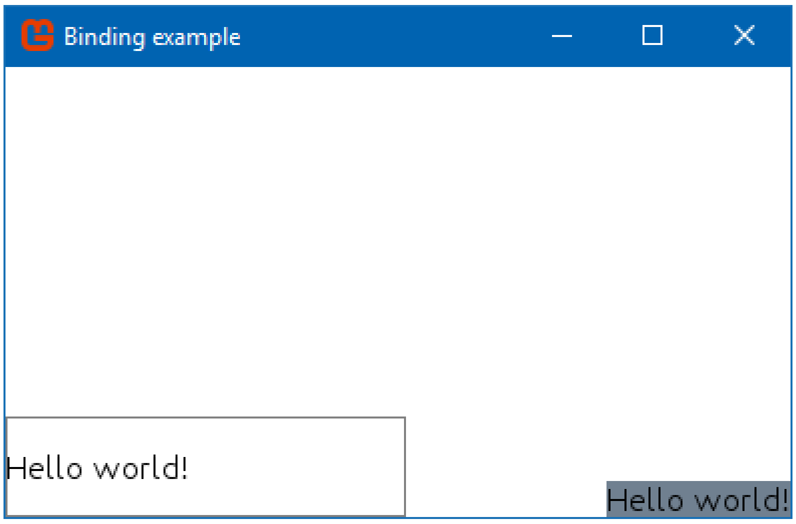


Figure 67: The result of the Binding example

Like other examples, also this example is a part of **Attachment F**. However, the included version of this example also contains a showcase of using an object implementing the `INotifyPropertyChanged` interface as a source for this binding.

5.4.5. More examples

All the above and even more examples are included as part of the **Attachment F**. The following are additional examples included in the mentioned attachment:

- Extended data binding example
- Dispatcher example
- Scrollable user interface example
- Creating custom scroll control example

As this library targets the full compatibility with the WPF API, more examples can be found on the appropriate page on the *Microsoft Developer Network*.

5.5. Features overview

In this section, we will provide an overview of all the features that have been implemented into the final version of this library as well as provide information about functionality with different behavior in comparison with the WPF API.

5.5.1. Dependency properties and objects

The system of dependency properties is implemented with the same API as the WPF provides. Dependency properties can be registered, read, and written to through the same methods as in the WPF. Support for creating read-only dependency properties is provided. The thread-safety of this system is assured. However, opposed to the WPF, this library allows the creation of dependency objects only on the main thread.

Dependency properties can register their `ValidateValueCallback` and also their `PropertyMetadata`. The system of overridable metadata for dependency properties is supported and deriving classes can provide their own metadata, which will take effective precedence.

5.5.2. Property metadata

The `PropertyMetadata` and `FrameworkPropertyMetadata` can be used to define certain behavior aspects of a dependency property.

Support is included for defining default values (which are also made unmodifiable), `PropertyChangedCallback`, and `CoerceValueCallback`.

In the case where `FrameworkPropertyMetadata` are used for a dependency property, there is also support for the following `FrameworkPropertyMetadataOptions` flags: `AffectsMeasure`, `AffectsArrange`, `AffectsRender`, `AffectsParentArrange`, and `AffectsParentMeasure`.

5.5.3. Data Binding

This library supports creation of data bindings through the static `BindingOperations` class. Those data bindings can be created either between two dependency objects or a dependency object and object. Like in the WPF, the dependency object must always be on the *Target* side of binding. Moreover, if the source object implements the `INotifyPropertyChanged` interface, data binding also supports automatic data updates.

The *Value Converters* are supported and so are the *OneWay* and *TwoWay* binding modes.

5.5.4. Freezables

The `Freezable` class is implemented in this library and contains functionality that enables cloning and freezing of deriving classes.

However, the cloning of expressions is not implemented and only value cloning can be performed. Therefore, whenever the `Clone` method of any `Freezable` object is called, it actually performs the work of the `CloneCurrentValue` method. The process of implementing the necessary functionality for these features in deriving classes is the same as in the WPF.

Support is also implemented for the `ReadPreamble` and `WritePreamble` methods that can be used to check whether the appropriate action can be performed on properties that are not dependency properties.

Finally, the `Freezable` class raises the `Changed` event whenever value of a dependency property is changed.

5.5.5. UIElement and FrameworkElement

All controls are based on the `UIElement`. This library implements the WPF way of creating controls and allows the developers to step into the layout process through `MeasureCore`, `ArrangeCore` method on the *core-level* layout and through `MeasureOverride`, and `ArrangeOverride` methods on the *framework-level* layout.

The `FrameworkElement` is implemented only partially, specifically this library provides only an implementation for its *framework-level* layout. Therefore, the following properties are taken into consideration on this layout level:

- `MinWidth, MinHeight`
- `MaxWidth, MaxHeight`
- `Width, Height`
- `VerticalAlignment`
- `HorizontalAlignment`

Any custom control can also make use of the `VisualOffset` property to position itself.

For the `UIElement` this library also supports the way of defining custom rendering instructions through its `OnRender` method. The provided `DrawingContext` includes support for the following rendering actions:

- `PushOpacity, PopOpacity`
- `DrawLine`
- `DrawRectangle`
- `DrawEllipse`
- `DrawRoundedRectangle`
- `DrawText`

The syntax for these methods, with the exception for `DrawText`, is the same as in the WPF. The `DrawText` only supports unformatted text and its parameters were altered to suit the needs of this library. There is also one method that is not defined in the WPF. The `PopOpacity` method is used to remove lastly applied `Opacity` for rendering instructions. In the WPF this is achieved by unified `Pop` method that removes any lastly pushed instruction that affects the rendering (like `Opacity`, `Effect`, or a `Transformation`).

As can be seen on the names of the mentioned `DrawingContext` methods, this library supports the rendering of the following graphical shapes:

- Line
- Rectangle
- Ellipse

- Rounded rectangle

For these shapes this library can render both their fill and their stroke. However, only a line and a rectangle can have a dashed stroke.

This stroke is being set up in the same way as in the WPF, through an instance of the `Pen` class. The `Pen` class supports currently only three properties: `Brush`, `Thickness`, and `DashStyle`. This library provides some premade dash styles that can be found as static resources in the static `DashStyles` class.

Finally, every `UIElement` inheritor can use the `VisualClip` property, that allows to set up a clipping (even non-rectangular) that is effective for the element itself and all its `Visual` children. Due to the way how this clipping is implemented the maximum number of nested clippings is limited to 255.

5.5.6. Controls

In this section we are going to go through the controls available in this library.

Layout controls

This library implements a version of the following controls: `Border`, `Canvas`, `Grid`, `ScrollContentPresenter`, `StackPanel`, `UserControl`.

The `Border` control provides support for `Padding` its content and is capable of drawing its border with differently thick sides.

The `Canvas` control allows to position its children based on the value of their `Canvas.Left` and `Canvas.Top` attached properties. Support for `Canvas.Right` and `Canvas.Bottom` is not currently provided by this library.

The `Grid` control provides support for positioning the children controls inside its available space. Support is included only for `RowDefinitions` and `ColumnDefinitions` with a fixed size.

The `ScrollContentPresenter` control is just an example of a scrollable control. It can hold an element and mouse wheel can be used to scroll through its content.

The `StackPanel` control can be used for stacking child elements either vertically or horizontally. Both of these orientations are supported.

The `UserControl` control defines the `Content` property that can be used while inheriting from the `UserControl` to define custom element structure for the new control.

Input controls

This library implements a basic version of the following input controls: `Button`, `TextBox`.

The `Button` control defines several visual properties that can be used to change its look, like in the WPF. It also provide `Click` event and allows any object as its content.

The `TextBox` control is used to enter text. This control is very basic and currently supports deleting current text using either *Backspace* or *Delete,* selecting text using mouse, and of course inserting a new text by writing on the keyboard.

Information controls

This library implements a basic version of the following information controls: `Image`, `Label`, `TextBlock`.

The `Image` control can be used to display an image. The source of an image is set through the `Source` property, that accepts a `Texture2D`, which is different than in the WPF.

The `Label` control can be used to show any content. For this it defines the `Content` property. This content can be any object (`ToString` is called and the result is shown inside the `Label`) or `UIElement` (in this case the content is rendered as part of the *Visual Tree*).

The `TextBlock` control is a simple control that is capable of displaying unformatted with defined color.

Selection controls

This library implements a basic version of the `CheckBox` selection control. This controls provides event notifications when checked and unchecked and allows to define its description through the `Content` property. This can be any `object`.

Shape controls

This library implements the `Shape`-based controls that are also supported for rendering by this library (as stated in section previous section 5.5.5). The implemented `Shape` controls include `Line`, `Ellipse`, and `Rectangle`.

The `Ellipse` and `Rectangle` also support the `Stretch` property.

5.5.7. Window

The Window class is used as the root element of the user interface. The `Title` property can be used to change the title of the MonoGame client window and its `ResizeMode` property can be used to define whether the MonoGame client window can be resized or not.

5.5.8. Routed events

This library implements the system of routed events. It supports registering new routed events, supports subscription to instanced events defined on a `UIElement` (even handled events). It is also possible to register *Class handlers* using the static

EventManager. Routed events can be manually raised by calling the RaiseEvent method of the UIElement.

Included is support for all three kinds of routing strategies – Bubble, Direct, and Tunnel. The included event system raises both, the *Non-Preview* and *Preview* types of input events.

The following list contains all supported input events that are being raised on UIElement-based objects:

- PreviewMouseMove | MouseMove
- PreviewMouseWheel | MouseWheel
- PreviewMouseDown | MouseDown
- PreviewMouseUp | MouseUp
- MouseEnter
- MouseLeave
- PreviewKeyDown | KeyDown
- PreviewKeyUp | KeyUp
- GotFocus
- LostFocus

Moreover, because the *Triggers* are not supported and the UIElement.IsMouseOver property is considered to be important during the creation of custom controls, this library implements a workaround so the developers get notified on value changes of this property. For this purpose, the UIElement contains a protected virtual OnIsMouseOverChanged method that is called by the event system whenever necessary.

Touch events and touch input are currently not supported by this library.

5.5.9. Brushes

This library implements three kinds of brushes:

- SolidColorBrush
- LinearGradientBrush
- ImageBrush

The SolidColorBrush supports defining a custom color or can use one of the predefined colors. As a source for these predefined colors, this library uses the static color resources defined in the Microsoft.Xna.Framework.Color struct. However, the WPF uses the Colors struct, which causes incompatibility with color definitions.

The LinearGradientBrush is used the same way as in the WPF and allows setting the StartPoint, EndPoint for a gradient, including a collection of gradient stops. However, the maximum number of gradient stops is limited to *12*.

The ImageBrush supports the Stretch property and to set the image source, it provides the ImageSource property. However, the type of this property is Texture2D, which is a break from the WPF API.

6 Conclusion

In this chapter, we will conclude this thesis. We will break this chapter into three parts. In the first part (6.1) we will sum up what was achieved in this library. In the second part (6.2) we will go through known issues at the final version of this library and finally in the last part (6.3) we will go through some ideas on how to enhance the functionality of this library in possible future versions.

6.1. Final results

In this section, we will assess the results of this thesis and compare it with the goals we established for this thesis in the section 1.4.

(G1) MonoGame GUI Library

This library is composed of one managed assembly for the MonoGame, and one content file containing a shader. The assembly does not have any platform-specific dependencies and the shader can be compiled for use with either the DirectX or the OpenGL. Therefore, there should be no serious problems with porting this library to other platforms.

(G2) Rendering

Included is support for rendering four different graphical shapes – line, rectangle, ellipse, and rounded rectangle. With all these primitives, we also support drawing their strokes. In the case of line and rectangle these can be dashed. Moreover, the rendering system was designed in a way so it can be easily extended with support for additional primitives. The primitives can also be textured or colored. For this, our library provides three different types of brushes: *SolidColorBrush*, *LinearGradientBrush*, and *ImageBrush*.

(G3) Event-driven user input

We provided an event-based input system based on the concept of *Routed events*. These events can be registered to travel through the user interface in either of the following ways: *Direct*, *Tunnel*, and *Bubble*. This library makes it also possible to subscribe to these events using either *Instanced handlers* or *Class handlers*. Exposed events include various keyboard and mouse events ranging from *KeyDown* to *MouseWheel* events.

(G4) Stock controls

The stock controls include various controls from each control area. There are two input controls: *Button* and *TextBox*. In the area of data presenting controls we provided the *Image*, *Label* and *TextBlock* controls. To support various positioning of our controls we implemented layout controls like the *Border*, *Canvas*, *Grid* and

StackPanel. There is also included a demonstrational version of a control providing scrolling functionality, the *ScrollContentPresenter*, and one selection control, the *CheckBox*. Finally, a *Shape* class based variants of our supported graphical shapes are implemented too.

(G5) Controls customizability

The customization of our stock controls is done through setting their public properties. This is mostly used to set a thickness of a border or the brush for various appearance properties like *Background* or *BorderBrush*.

(G6) Custom controls

What our basic stock controls cannot do can be easily achieved by the developers by creating their own custom controls. To make this process most easy and feature-rich our library implements the layout system of the WPF, including the ability to affect the *Measure* and *Arrange* layout process either on *UIElement* or *FrameworkElement* level. To provide custom rendering instructions for any custom element we implemented the `OnRender` method. This method provides *DrawingContext* that is capable of rendering all the graphical shapes mentioned at the beginning of this chapter. A *VisualClip* property is also made available to clip the content of a custom control and its children. This property supports clipping using all the geometrical shapes we are able to render and can be also used by any third party developer to implement a custom version of scrolling control.

(G7) WPF API

Finally, all the functionality implemented in this library follows strictly the WPF API with only a minor variations to enable easy code porting and make it possible for the future versions of this library to continue implementing the WPF API.

On the following two figures, we can see the same user interface code running on the WPF (Figure 68) and on this library (Figure 69). The solutions for both these applications are included among the examples as part of **Attachment F**.

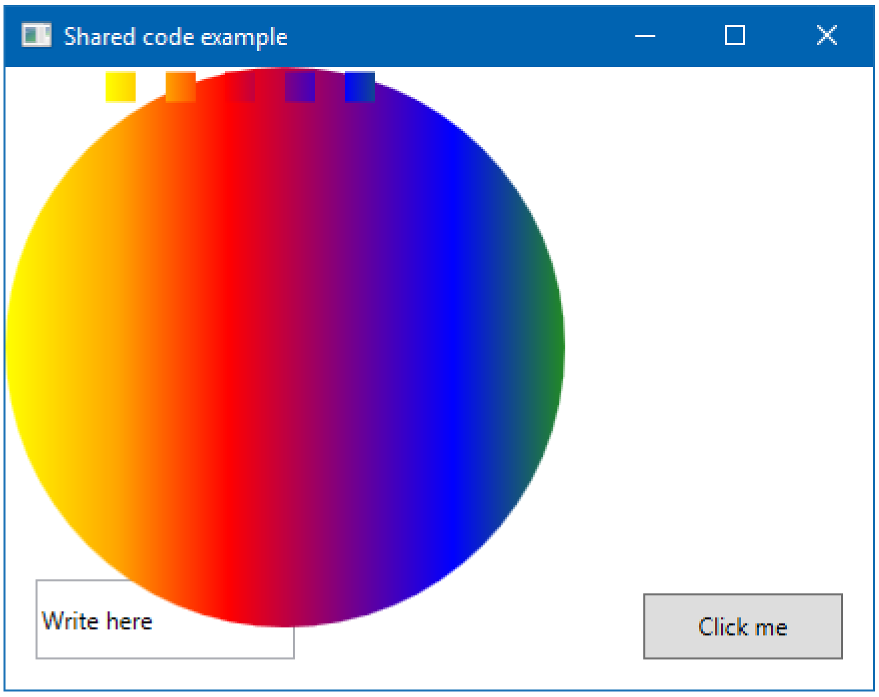


Figure 68: The result of the same user interface code. Code running on the WPF.

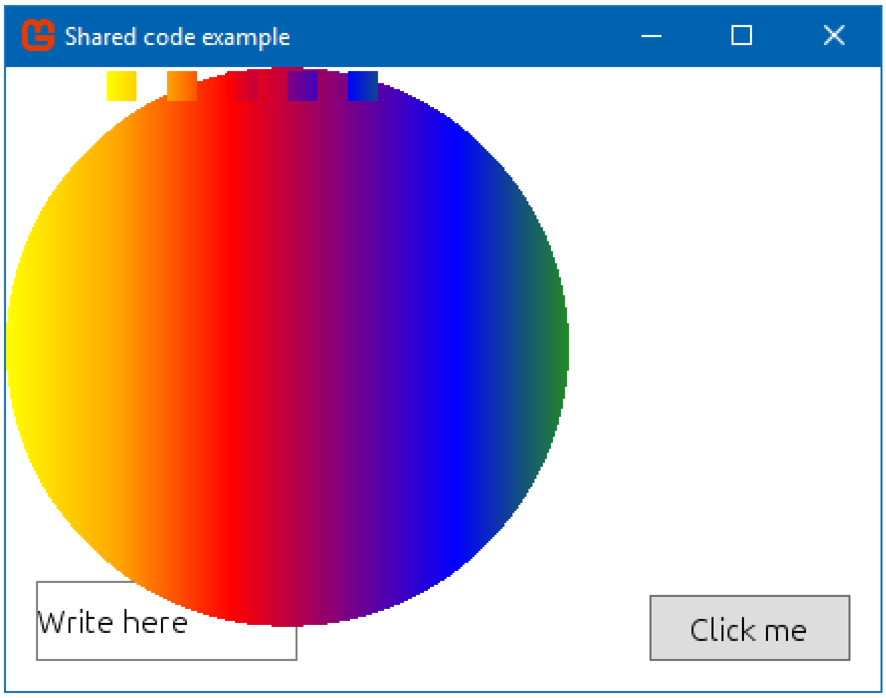


Figure 69: The result of the same user interface code. Code running on this library.

Additional features

On top of those features requested by our goals, we also implemented additional features like support for *OneWay* and *TwoWay Data bindings*, and cloning and freezing objects that derive from the *Freezable* class.

Evaluation of goals

When we compare our library with the goals mentioned in the chapter 1.4, we can conclude we have fulfilled all of them.

6.2. Known issues

The final version of this library also have some known issues. Below is the list of the most notable ones.

- When an opacity is applied to a parent control, this opacity is also applied during rendering to individual children and their own rendering. This causes any overlapping children elements to alpha blend. This is however in contrast with the WPF that seems to render the children first and apply the opacity for all this rendered content in the end. Possible solution for this issue is to use a temporary `RenderTarget2D` that will contain all the rendered children and when the rendering of child elements is finished, this `RenderTarget2D` would be drawn with the set opacity.
- Graphical shapes rendered by this library have aliasing problems. MonoGame includes an anti-aliasing solution, however this is available only for OpenGL platforms and not the DirectX [47]. A solution for this would be to create a custom shader that would implement some anti-aliasing technique like *FXAA* [48] or *MSAA* [49] and would be applied while rendering the user interface. This unified approach could be used to enable anti-aliasing on both the DirectX and the OpenGL platforms.
- Changes in sub-properties of an object that is set as a value for a dependency property do not cause controls deriving from the `UIElement` to be invalidated (as set in the `FrameworkPropertyMetadataOptions` flags for the dependency property). Because this functionality seems to be implemented only for the `Freezable`-based dependency properties, possible fix for this issue would be to subscribe for the `Changed` event whenever a `Freezable` object is set as a value for a dependency property.
- The stroke of the rounded rectangle is not rendered correctly under circumstances when the `RadiusX` or `RadiusY` property is smaller than the requested thickness of the stroke.
- When an empty `TextBox` is focused by mouse and some text is typed in, it is necessary to click once again somewhere on the `TextBox`. Otherwise, an

unhandled exception is thrown when trying to delete a text using either Backspace or Delete key.

6.3. Future Work

While the library matches our goals, it is still merely a subset of functionality the WPF provides. Therefore, this still represents many features left to implement. We will mention only those that would be of highest benefit at this point. Moreover, there are some areas of the library, where the functionality can be improved.

- During the work on this library, main emphasis was on implementing the core functionality of the library. This was to provide the developers a powerful way to create their custom controls. This lead to the fact that the included controls in this library are basic and lack some functionality. Therefore, now when the core of the library is finished, more work would be needed on the included controls. Including the possibility of implementing additional controls.
- The library currently implements only four different graphical shapes. There are however many more in the WPF. Implementing these additional shapes would provide more ways to developers how they can customize their user interfaces.
- The library does not offer a quick way to affect the rendering of a shape geometry. By implementing support for *Transformations* and *RenderTransforms* we would be able to provide the developers with a simple to manipulate computed geometry without forcing the library to recompute it.
- If the *Transformations* are to be supported, all geometries generated for use as *LineList* primitives (currently the stroke of ellipse and rounded rectangle) need to be changed to be generated either as the *TriangleList* or *TriangleStrip* primitives, otherwise they might not retain the intended geometry.
- All the rendering operations are handled by the `RenderContext` class, this makes it possible for us to consider the possibility of completely abstracting the rendering process. This would allow this library to run on many different rendering back-ends.
- Support for *Animations* could be implemented to make the user interfaces look more alive.
- This library works on the level of individual pixel units, this can be however problematic for someone attempting to use this library on a modern smartphone. The density of the pixels is enormous on these devices, and this would cause the user interfaces rendered by this library to be way too small for any reasonable interaction. Therefore, support for rendering in *Device independent pixels* [4846] would make this library more usable.
- In the Analysis chapter, we examined the possibility of using a vector font for our library (section 3.4) and in the end decided that we will use the standard bitmap-based font. This vector font would be a good addition for any future

version of this library as it would allow not only to directly use fonts that are already present with the operating system but would also provide a way to produce scalable and transformable high quality text.

- Some implementation of *Style* or *Template* system is needed to provide more advanced control customization support for individual controls.

7 Attachments

This thesis comes with a CD that includes the following attachments.

Attachment A: Source code

The Visual Studio solution for this library can be found in the `/MonoGameWPF` directory. This solution contains the `MonoGameWPF` project with the source code of the library.

Attachment B: Documentation

The documentation is located in the `/Documentation` directory and is made from the source code using the Sandcastle Help File Builder project [51].

Attachment C: MonoGame setup file

The setup file for the MonoGame 3.4 is located in the `/MonoGameSetup` directory.

Attachment D: Visual Studio Template

The template file for the Visual Studio is included in the `/Template` directory.

Attachment E: Compiled library for Windows DirectX platform

The compiled managed assembly of this library along the compiled shader file and two included fonts can be found in the `/Compiled` directory.

Attachment F: Solutions with code examples

Examples of user interfaces showcased in the chapter 5.4, along with some others not mentioned there, are located in the form of individual solutions in the `/Examples` directory.

8 References

1. Minecraft. Retrieved December 3, 2015, from https://minecraft.net/
2. Terraria. Retrieved December 3, 2015, from https://terraria.org/
3. Write once, play everywhere. Retrieved December 3, 2015, from http://www.monogame.net/
4. GUI Library for MonoGame. Retrieved December 3, 2015, from http://gamedev.stackexchange.com/questions/42142/gui-library-for-monogame
5. XWinForms. Retrieved December 3, 2015, from http://sourceforge.net/projects/xwinforms/
6. IONSTAR Studios. Retrieved December 3, 2015, from http://www.ionstar.org/
7. Bitbucket. Retrieved December 3, 2015, from https://bitbucket.org/sparklinlabs/nuclearwinter
8. Ruminate MonoGame GUI. Retrieved December 3, 2015, from http://xnagui.codeplex.com
9. Nuclex Framework. Retrieved December 3, 2015, from http://nuclexframework.codeplex.com
10. MonoGame Gui4U. Retrieved December 3, 2015, from https://gui4u.codeplex.com
11. Coherent Labs. Retrieved December 3, 2015, from http://coherent-labs.com
12. Apple Lisa. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Apple_Lisa. Image from https://upload.wikimedia.org/wikipedia/en/5/52/Apple_Lisa_Office_System_3.1.png
13. Warcraft III: Reign of Chaos. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Warcraft_III:_Reign_of_Chaos. Image from https://upload.wikimedia.org/wikipedia/en/f/fa/Reign_of_Chaos_campaign.png
14. Common WPF Controls. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/bb655881(v=vs.90).aspx
15. Routed Events Overview. Retrieved December 3, 2015, from https://msdn.microsoft.com/library/ms742806(v=vs.100).aspx. Image from https://i-msdn.sec.s-msft.com/dynimg/IC130301.png
16. UIElement.Clip Property. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/system.windows.uielement.clip(v=vs.110).aspx. Image from https://i-msdn.sec.s-msft.com/dynimg/IC167464.png
17. Mono. Retrieved December 3, 2015, from http://www.mono-project.com
18. Mono on WPF. Retrieved December 3, 2015, from http://www.mono-project.com/docs/gui/wpf/
19. Microsoft Silverlight. Retrieved December 3, 2015, from https://www.microsoft.com/silverlight/

20. Mono (Moonlight). Retrieved December 3, 2015, from http://www.mono-project.com/docs/web/moonlight/
21. Cairo. Retrieved December 3, 2015, from http://cairographics.org/
22. Primitive Topologies included in DirectX 10. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx
23. OpenGL Primitive. Retrieved December 3, 2015, from https://www.opengl.org/wiki/Primitive
24. How can I draw a simple 2D line in XNA without using 3D primitives and shders. Retrieved December 3, 2015, from http://gamedev.stackexchange.com/questions/44015/how-can-i-draw-a-simple-2d-line-in-xna-without-using-3d-primitives-and-shders
25. Draw Rectangle in XNA using SpriteBatch. Retrieved December 3, 2015, from http://stackoverflow.com/questions/5751732/draw-rectangle-in-xna-using-spritebatch
26. SpriteBatch class source code. Retrieved December 3, 2015, from https://github.com/mono/MonoGame/blob/develop/MonoGame.Framework/Graphics/SpriteBatch.cs
27. SpriteBatcher class source code. Retrieved December 3, 2015, from https://github.com/mono/MonoGame/blob/develop/MonoGame.Framework/Graphics/SpriteBatcher.cs
28. Morrill, J. (2011, February 13). A Critical Deep Dive into the WPF Rendering System. Retrieved December 3, 2015, from https://jeremiahmorrill.wordpress.com/2011/02/14/a-critical-deep-dive-into-the-wpf-rendering-system
29. Flow Control Limitations. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/windows/desktop/bb219848(v=vs.85).aspx
30. Petzold, C. (2015, March 1). DirectX Factor: Triangles and Tessellation. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/magazine/dn605881.aspx
31. Vertex Buffer Object. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Vertex_Buffer_Object
32. FreeType project. Retrieved December 3, 2015, from http://www.freetype.org/
33. SharpFont library. Retrieved December 3, 2015, from https://github.com/Robmaister/SharpFont
34. Scissor Test. Retrieved December 3, 2015, from https://www.opengl.org/wiki/Scissor_Test
35. Rendering Pipeline Overview. Retrieved December 3, 2015, from https://www.opengl.org/wiki/Rendering_Pipeline_Overview
36. Stencil Test. Retrieved December 3, 2015, from https://www.opengl.org/wiki/Stencil_Test

37. Hit Testing in the Visual Layer. Retrieved December 3, 2015, from https://msdn.microsoft.com/library/ms752097(v=vs.100).aspx. Image from https://i-msdn.sec.s-msft.com/dynimg/IC9903.png
38. Attached Properties Overview. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/vstudio/ms749011(v=vs.100).aspx
39. Implementing Finalize and Dispose to Clean Up Unmanaged Resources. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.100).aspx
40. Point in triangle test. Retrieved December 3, 2015, from http://www.blackpawn.com/texts/pointinpoly/
41. Two-Point Form of line equation. Retrieved December 3, 2015, from http://mathworld.wolfram.com/Two-PointForm.html
42. Bézier curve. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/B%C3%A9zier_curve
43. LinearGradientBrush Class. Retrieved December 3, 2015, from https://msdn.microsoft.com/en-us/library/system.windows.media.lineargradientbrush(v=vs.110).aspx. Image from https://i-msdn.sec.s-msft.com/dynimg/IC143073.jpeg
44. Getting Effect .fx files to compile and run Hints, Tips and Gotchas. Retrieved December 3, 2015, from https://github.com/mono/MonoGame/wiki/Getting-Effect-.fx-files-to-compile-and-run---Hints,-Tips-and-Gotchas
45. Scalar Projection. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Scalar_projection
46. Ubuntu Font Family. Retrieved December 3, 2015, from http://font.ubuntu.com/
47. DirectX can't enable Anti Aliasing. Retrieved December 3, 2015, from https://github.com/mono/MonoGame/issues/3571
48. FXAA. Retrieved December 3, 2015, from http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
49. Multisample anti-aliasing. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Multisample_anti-aliasing
50. Device independent pixel. Retrieved December 3, 2015, from https://en.wikipedia.org/wiki/Device_independent_pixel
51. Sandcastle Help File Builder project. Retrieved December 3, 2015, from https://github.com/EWSoftware/SHFB